An Unsolicited Soliloquy on Dependency Parsing

Mark Anderson

tese de doutoramento, 2021

Programa de Doutoramento en Computación

Director: Carlos Gómez Rodríguez

Universidade da Coruña

Dr. Carlos Gómez Rodríguez, Associate Professor at the Department of Computer Science and Information Technologies at the Universidade da Coruña,

certifies

that this dissertation titled "An Unsolicited Soliloquy on Dependency Parsing", submitted to Universidade da Coruña by Mark Anderson, has been carried out under my supervision and meets all the requirements for the award of *Ph.D. Degree*.

Dr. Carlos Gómez Rodríguez, Prof. Titular de Universidad en el Departamento de Ciencias de la Computación y Tecnologías de la Información de la Universidade da Coruña

certifica

que esta tesis titulada "An Unsolicited Soliloquy on Dependency Parsing", depositada en la Universidade da Coruña por Mark Anderson, se ha llevado a cabo bajo mi supervisión y cumple los requisitos para optar al grado de *Doctor*.

Dr. Carlos Gómez Rodríguez
Depto. de Ciencias de la Computación y Tecnologías de la Información
Universidade da Coruña

Acknowledgements

This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (FASTPARSE, grant agreement No 714150) and from the Centro de Investigación de Galicia (CITIC) which is funded by the Xunta de Galicia and the European Union (ERDF - Galicia 2014-2020 Program) by grant ED431G 2019/01.

Abstract

This thesis presents work on dependency parsing covering two distinct lines of research. The first aims to develop efficient parsers so that they can be fast enough to parse large amounts of data while still maintaining decent accuracy. We investigate two techniques to achieve this. The first is a cognitively-inspired method and the second uses a model distillation method. The first technique proved to be utterly dismal, while the second was somewhat of a success.

The second line of research presented in this thesis evaluates parsers. This is also done in two ways. We aim to evaluate what causes variation in parsing performance for different algorithms and also different treebanks. This evaluation is grounded in dependency displacements (the directed distance between a dependent and its head) and the subsequent distributions associated with algorithms and the distributions found in treebanks. This work sheds some light on the variation in performance for both different algorithms and different treebanks. And the second part of this area focuses on the utility of part-of-speech tags when used with parsing systems and questions the standard position of assuming that they might help but they certainly won't hurt.

Resumen

Esta tesis presenta trabajo sobre análisis de dependencias que cubre dos líneas de investigación distintas. La primera tiene como objetivo desarrollar analizadores eficientes, de modo que sean suficientemente rápidos como para analizar grandes volúmenes de datos y, al mismo tiempo, sean suficientemente precisos. Investigamos dos métodos. El primero se basa en teorías cognitivas y el segundo usa una técnica de destilación. La primera técnica resultó un enorme fracaso, mientras que la segunda fue en cierto modo un éxito.

La otra línea evalúa los analizadores sintácticos. Esto también se hace de dos maneras. Evaluamos la causa de la variación en el rendimiento de los analizadores para distintos algoritmos y corpus. Esta evaluación utiliza la diferencia entre las distribuciones del desplazamiento de arista (la distancia dirigida de las aristas) correspondientes a cada algoritmo y corpus. También evalúa la diferencia entre las distribuciones del desplazamiento de arista en los datos de entrenamiento y prueba. Este trabajo esclarece las variaciones en el rendimiento para algoritmos y corpus diferentes. La segunda parte de esta línea investiga la utilidad de las etiquetas gramaticales para los analizadores sintácticos.

Resumo

Esta tese presenta traballo sobre análise sintáctica, cubrindo dúas liñas de investigación. A primeira aspira a desenvolver analizadores eficientes, de maneira que sexan suficientemente rápidos para procesar grandes volumes de datos e á vez sexan precisos. Investigamos dous métodos. O primeiro baséase nunha teoría cognitiva, e o segundo usa unha técnica de destilación. O primeiro método foi un enorme fracaso, mentres que o segundo foi en certo modo un éxito.

A outra liña avalía os analizadores sintácticos. Esto tamén se fai de dúas maneiras. Avaliamos a causa da variación no rendemento dos analizadores para distintos algoritmos e corpus. Esta avaliación usa a diferencia entre as distribucións do desprazamento de arista (a distancia dirixida das aristas) correspondentes aos algoritmos e aos corpus. Tamén avalía a diferencia entre as distribucións do desprazamento de arista nos datos de adestramento e proba. Este traballo esclarece as variacións no rendemento para algoritmos e corpus diferentes. A segunda parte desta liña investiga a utilidade das etiquetas gramaticais para os analizadores sintácticos.

Publications

Covered in thesis

1. Mark Anderson, David Vilares, and Carlos Gómez-Rodríguez. 2019. Artificially evolved chunks for morphosyntactic analysis. In *Proceedings of the 18th International Workshop on Treebanks and Linguistic Theories (TLT, SyntaxFest 2019)*, pages 133–143, Paris, France. Association for Computational Linguistics

2. Mark Anderson and Carlos Gómez-Rodríguez. 2020b. Inherent dependency displacement bias of transition-based algorithms. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 5147–5155, Marseille, France. European Language Resources Association

3. Mark Anderson and Carlos Gómez-Rodríguez. 2020c. On the frailty of universal POS tags for neural UD parsers. In *Proceedings of the 24th Conference on Computational Natural Language Learning*, pages 69–96, Online. Association for Computational Linguistics

4. Mark Anderson and Carlos Gómez-Rodríguez. 2020a. Distilling neural networks for greener and faster dependency parsing. In *Proceedings of the 16th International Conference on Parsing Technologies and the IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies*, pages 2–13, Online. Association for Computational Linguistics

5. Mathieu Dehouck, Mark Anderson, and Carlos Gómez-Rodríguez. 2020. Efficient EUD parsing. In *Proceedings of the 16th International Conference on Parsing Technologies and the IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies*, pages 192–205, Online. Association for Computational Linguistics

6. Mark Anderson and Carlos Gómez-Rodríguez. 2021. What taggers fail to learn, parsers need the most. In *Proceedings of the 23rd Nordic Conference of Computational Linguistics (NoDaLiDa 2021)*, Online

Not covered in thesis

7. Mark Anderson, Carlos Gómez Rodríguez, and Anders Søgaard. 2021. Replicating and extending "Because their treebanks leak": Graph isomorphism, covariants, and parser performance. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics*, Online. Association for Computational Linguistics

8. Yova Kementchedjhieva, Mark Anderson, and Anders Søgaard. 2021. John praised Mary because *he*? Implicit causality bias and its interaction with explicit cues in LMs. In *Findings of the Association for Computational Linguistics: ACL 2021*, Online. Association for Computational Linguistics

Introduction

This thesis is a hodgepodge of work with a central theme. It is based on a collection of papers that I have brought together to form something somewhat coherent. However, it is a thesis not a novel, so I don't even believe that it really requires a strong narrative structure. The *theme* of the thesis is dependency parsing. It is common to justify working in a subfield of NLP by highlighting how the subfield in question can help other subfields of NLP or other artificial intelligence tasks. I personally don't think that is necessary. Dependency parsing is an interesting pursuit in its own right.

Dependency parsing is the act of casting linguistic data into a formal syntactic structure, specifically a dependency tree (or graph) as defined by a dependency grammar. Dependency grammar and dependency parsing are formally introduced in Chapter 1. In Chapter 2, you will find a general coverage of recent work in dependency parsing, covering recent trends and the current state of affairs as they relate to this thesis.

The novel work presented here falls into two bins quite nicely. The first covers attempts to actually develop systems for dependency parsing. This work was done in the FastParse group. And it was an apt name because we tried to develop techniques and systems that were efficient. Efficiency in this context meant being fast (at inference time) and still somewhat accurate. The first technique we tried was influenced by psycholinguistics. It is based on the working memory restrictions of the human brain and the hypothesised need for humans to create abstract hierarchical representations of linguistic input. This is done so that we don't lose information during the real-time processing of linguistic input. This is called Chunk-and-Pass processing. Chapter 3.4 covers our attempts to use this to make parsers faster while maintaining accuracy. It was an abject failure, but some of the auxiliary work that was done to implement this has the potential to be useful and interesting. The second method we used to develop efficient parsers was more of a success. Fundamentally, we used larger models with more parameters to help guide models with less parameters (therefore faster) using a model distillation technique. This resulted in the fastest modern parsing system which is also more accurate than the next fastest parsing system. This work is covered in Chapter 4.

The second part of the thesis focuses on evaluating parsers. Chapter 5 contains work focused on the dependency displacement of tokens in a sentence. This is just the directed distance between a token and its head (the word it depends on). We use this measurement as the basis of two similar analyses. The first compares the inherent distribution of dependency distances that certain transition-based algorithms are biased towards to those found in treebanks. We then observe a relationship between the similarity of these distributions and the performance of algorithms. In a similar vein, we compare the distributions of training and test data and find a strong correlation to their similarity and parsing performance even when accounting for covariants and for two parsing system paradigms. In Chapter 6, we evaluate part-of-speech tags and how the accuracy of taggers impacts the usefulness of these tags for parsing systems. It offers a thorough evaluation for both a graph-based and a transition-based parser and finds that predicted tags typically harm performance when compared to not using

any tags at all. We extend this analysis to evaluate if parsers learn anything intrinsically about word types which makes the information from these tags redundant or potentially conflicting. We obtain results that suggest parsers do learn something in this direction and that what they don't learn is also what taggers fail to capture. We also extend this analysis to look specifically at the usefulness of part-of-speech tags in low-resource contexts based on findings of the original analysis. Here we observe that smaller treebanks can more readily leverage something from predicted tags even when the accuracy of the taggers is not particularly high.

The work throughout is multilingual. We use Universal Dependency treebanks and the number of languages covered in a given experiment range from 4 to 80 (or thereabouts). We also use a range of parsing paradigms, so that's another multi-facet. I would like to stress that dependency parsing is interesting in its own right. Whether it can aid a neural translator or some gargantuan language model is certainly an interesting question, but it isn't one I sought to answer here.

Contents

List of Figures

List of Tables

List of Algorithms

List of Examples

PART I

Dependency Parsing

Chapter 1

Preliminaries

In this chapter we give a brief introduction to dependency grammar (Section 1.1) and dependency parsing (Section 1.2). We then introduce the dependency grammar framework predominately used throughout this thesis (Section 1.3) and give a brief overview of some alternative frameworks. Finally, Section 1.4 describes common systems used for dependency parsing. The details presented here should be sufficient for understanding the subsequent chapters, but each subsequent chapter should be understandable independent of one another so they can be read in any order or combination.

1.1 Dependency grammar

Dependency grammar is a set of syntactic representations that spans a number of frameworks and theories. Its modern form originated in the work of Tesnière (1959). Fundamental to all varieties of dependency grammar is the concept that the syntactic structure of a linguistic instance (sentence) is encoded by asymmetric and binary relations between the elements (typically tokens) of the instance, resulting in a dependency graph (typically a well-formed tree). The two tokens linked by each of the edges in this dependency tree are usually referred to as the *dependent* and the *head*, where the *dependent* is syntactically conditioned by the *head*. Figure 1.1 shows an example of a dependency tree as it would be encoded in the Universal Dependencies formalism. Different theories and frameworks use different criteria to decide how to connect the elements.

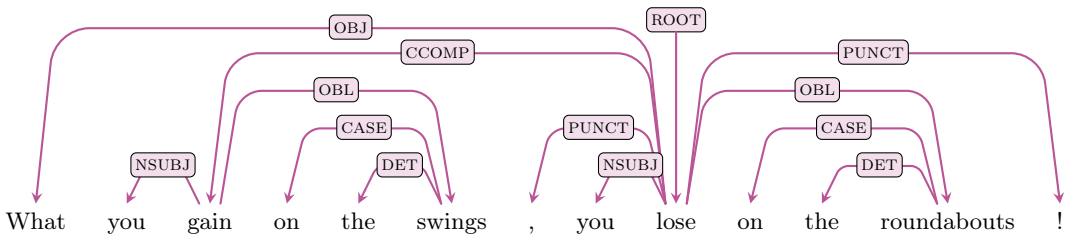
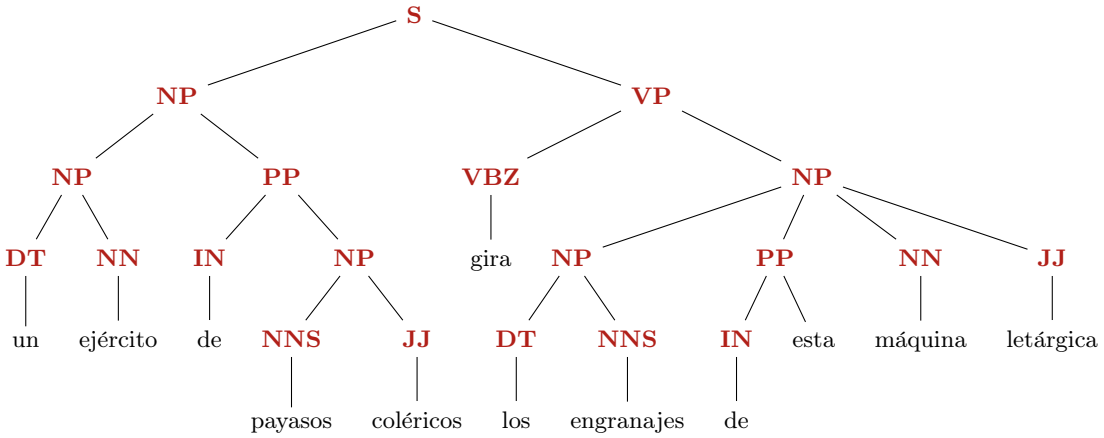


Figure 1.1: Dependency tree example.

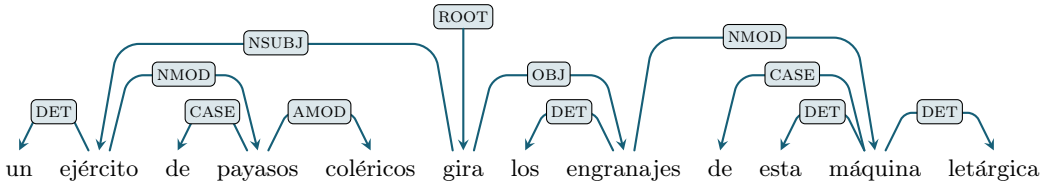
1.1.1 Constituency grammar

A common alternative grammar formalism is constituency grammars. Constituency grammars are a set of frameworks which model syntax with hierarchical representations built on phrases or constituencies (Chomsky, 1957). The leaves of the tree are the tokens in a sentence and the inner nodes are non-terminal nodes which represent some phrase structure from a given

grammar, e.g. an inner node could be VP representing a verbal phrase which itself contains other non-terminal nodes such a noun phrase NP or a verb V. Figure 1.2 shows a comparison between a dependency tree and a constituency tree for the same sentence.



(a) Constituency tree



(b) Dependency tree

Figure 1.2: An example constituency tree (a) and the corresponding dependency tree (b).

1.2 Dependency parsing

Dependency parsing is the process of establishing the dependency graph ($\mathcal{G}$) of a given linguistic instance, typically a sentence ($\mathbf{x}$). Where a sentence is a series of tokens:

$$\mathbf{x} = w_1, w_2 ... w_l \tag{1.1}$$

where w_i is a token at point i of $\mathbf{x}$ and l is the length of $\mathbf{x}$. For any given formalism there exists a set of dependency relations such that:

$$R = \{r_1, r_2, ..., r_m\} \tag{1.2}$$

where m is the dimension of the relation space. A labelled graph $\mathcal{G}$ is then defined as a set of nodes and a corresponding set of edges:

$$\mathcal{G} = (\mathcal{N}, \mathcal{E}) \tag{1.3}$$

where $\mathcal{N}$ is a set of nodes ν corresponding to the words in $\mathbf{x}$ and $\mathcal{E}$ is the set of labelled directed edges ϵ connecting the nodes such that $\mathcal{E} \subseteq \mathcal{N} \times R \times \mathcal{N}$ such that an edge is given by:

$$\epsilon = (w_h, r, w_d) \tag{1.4}$$

where w_h is the head, w_d is the dependent, and r is the syntactic relation. Then for a given sentence $\mathbf{x}$ there exists a labelled directed graph $\mathcal{G}$ that correctly captures its syntax.[1] $\mathbb{S}$ is

[1]Note that it is possible that more than one parse is correct for a given sentence, but we assume there is one unique correct parse based on the annotation in the data.

then the set of all possible sentences and $\mathbb{G}$ is the set of all possible graphs, such that for a sample of data $\mathbb{D}$ consisting of a subset of $\mathbb{S}$ there exists a function f that maps the data from $\mathbb{D}$ to $\mathbb{G}$:

$$f(\mathbf{x}_i) = \mathcal{G}_i \text{ for } \mathbf{x}_i \in \mathbb{D},\ \mathcal{G}_i \in \mathbb{G} \tag{1.5}$$

Dependency parsing is therefore the approximation of this function f given a sample of data $\mathbb{D}$ such that the difference between each predicted graph $\mathcal{G}_p$ and its corresponding true graph $\mathcal{G}$ is minimised. The system that approximates this function is then called a dependency parser and is then used to parse new instances.

Different formalisms follow different guidelines, but typically dependency grammar formalisms enforce well-formed trees rather than unrestricted graphs. The first restriction is that for every token there exists only one incoming edge, i.e. each token has a single head. Second the trees must be acyclic such that if w_i is headed by w_j there is no path leading from w_i that ends at w_j, i.e. the tree contains no cycles. Third a tree is fully connected such that each pair of words is connected via a path when direction is ignored. So for each word in a sentence a dependency parser needs to predict one incoming labelled edge. Typically w_0 is a dummy node added so that the root of a sentence is headed by this node. So dependency parsing simplifies to the task of predicting each token's head and the corresponding relation label associated with each edge. For a deeper treatment of the formalism of dependency parsing see Kübler et al. (2009).

1.3 Universal Dependencies

We give a detailed description of the Universal Dependencies framework rather than other frameworks because the vast majority of the work presented in this thesis uses treebanks from this framework (McDonald et al., 2013). The main reason for using this framework is that it is inherently multilingual. Universal Dependencies (UD) is a multilingual enterprise seeking to develop a treebank annotation framework with crosslingual consistency. It is based on the universal Stanford dependencies annotation framework (de Marneffe et al., 2014). The UD framework works under a number of guiding principles that sometimes can conflict. This mainly manifests in the need for suitable linguistic analysis of phenemona found in a given language and the need for crosslingual consistency so as to be suitable for linguistic typology. Therefore UD considers words as the fundamental unit of syntax, which is sensible as NLP typically works on tokenised input, and so this makes UD more readily used in NLP systems. It also aids typological considerations, although Haspelmath (2011) argues that the definition of a word in a crosslingual sense is hard to pin down (or impossible) and this has been highlighted specifically as to how Japanese tokenisation is handled in UD, namely preferring short unit words to long unit words or *bunsetsu* which are more commonly considered *words* or at least the fundamental unit of syntax in Japanese (Murawaki, 2019). Based on the driving philosophy of having a framework that is suitable for linguistic typology, UD has opted to consider content and not function words to have syntactic primacy. This is mainly due to the wide difference in which the grammatical roles of function words are encoded in some languages, i.e. in some languages these roles are not expressed at all and in others they are morphologically expressed. For example, in English

Russian	дом	not marked
Basque	etxe**a**	definiteness marked with suffix
English	**the** house	definiteness marked with determiner

Example 1: How definiteness of a noun is marked in different languages as an example of how the grammatical role of certain function words (here the definite article in English) are encoded in different languages. In Russian it is not marked at all, whereas in Basque it is marked by a suffix (**a**) and in English it is marked by a determiner (**the**).

definiteness is marked by a determiner as in *the house*, whereas in Basque the same role is encoded with a suffix, and in Russian it isn't encoded at all (see Example 1).

The annotation scheme also consists of a universal set of part-of-speech (POS) tags based on those introduced by Petrov et al. (2012) and a universal set of morphosyntactic features of Zeman (2008). The UD framework consists of 37 dependency relations, 17 universal POS tags, and 24 morphological feature categories (7 lexical features, 7 nominal inflectional features, and 10 verbal inflectional features). The format of the data then comes in a revised form of the CoNLL-X structure (Buchholz and Marsi, 2006). This consists of 10 columns of: (1) the word index, (2) the word form, (3) the lemma/stem of the word, (4), the universal POS tag, (5) the language-specific POS tag (if available), (6) list of morphological features from the UD framework set of tags if available or sometimes a language-specific set, (7) the head index, (8) the UD relation, (9) the enhanced dependency graph information (if avialable), and (10) any other miscellaneous annotation. A full CoNLL-U instance taken from the Welsh-CCG treebank in UD v2.7 (Heinecke and Tyers, 2019) is given in Example 2 with the corresponding dependency tree shown in Figure 1.3. For more details see the Universal Dependencies website.[2]

ID	Form	Lemma	UPOS	XPOS	Feats	Head	Deprel	Deps	Misc
1	Maen	bod	VERB	verb	Mood=Ind|Num=Plur|Person=3|Tense=Pres	0	root	_	_
2	nhw	hwy	PRON	indep	Number=Plur|Person=3|PronType=Prs	1	nsubj	_	SA=No
3	'n	yn	AUX	impf	_	4	aux	_	_
4	mynd	mynd	NOUN	verbnoun	Number=Sing|VerbForm=Vnoun	1	xcomp	_	_
5	i	i	ADP	prep	_	6	case	_	_
6	Dregaron	Tregaron	NOUN	noun	Gender=Masc|Mutation=SM|Number=Sing	4	obl	_	_
7	!	!	PUNCT	punct	_	1	punct	_	SA=\n

Example 2: CoNLL-U instance corresponding to the tree (from Cymraeg Corpws Cystrawennol y Gymrae in UD v2.7) given in Figure 1.3. English: *They went to Tregaron.*

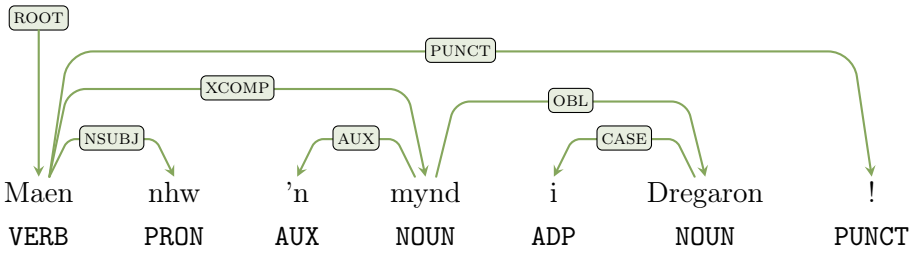


Figure 1.3: Dependency tree with UPOS tags corresponding to CoNLL-U data format instance shown in Example 2. English: *They went to Tregaron.*

1.3.1 Enhanced Universal Dependencies

The choices made in the UD framework have some undesired consequences such as long paths between related content words, which can obscure how these words are related and certain dependency relations are used in a number of different contexts which can limit how informative they are. An enhanced UD representation was therefore developed which encodes the syntax in a dependency graph rather than a tree. This allows for the propagation of edges to conjuncts, for explicit relations between subjects and embedded verbs, for a more nuanced handling of relative pronouns by attaching the referent as a dependent of the main predicate of the relative clause and the pronoun as a dependent of the referent, and for the handling of ellipsis. The relation labels are also enhanced to include relation subtypes and case information. A more in-depth description is given in Section 4.5 where this framework is used.

[2] https://universaldependencies.org/guidelines.html

1.3.2 Surface-Syntactic Universal Dependencies

Osborne and Gerdes (2019) argue that the primacy of content words over function words is not linguistically sound. The claim is based on the semantic aspect of the distinction between content and function words which muddies the water with respect to analysing syntactic phenomena with UD. Gerdes et al. (2018) therefore developed Surface-Syntactic Universal Dependencies (SUD). UD trees can be automatically converted to SUD with miminal human oversight and all treebanks available in the UD framework are also availabe in SUD.[3] SUD adheres to dependency grammar traditions by subordinating content words, e.g. adpositions are the head of the nouns they are associated with and auxiliary verbs head the content verbs they are associated with. A comparison of these two frameworks is shown in Figure 1.4 where you can see the auxiliary verb *est* heads *codée* in the SUD tree whereas it is the opposite in the UD tree. Also, the prepositions *sous* and *d'* (contracted form of *de*) head their respective nouns *forme* and *ADN* (DNA). We mention this framework because it is interesting and also to highlight that the results presented in this thesis are true for UD parsing and only UD parsing (although there is no strong reason why they should not extend to other frameworks) so we try to adopt a tempered approach in reporting results.

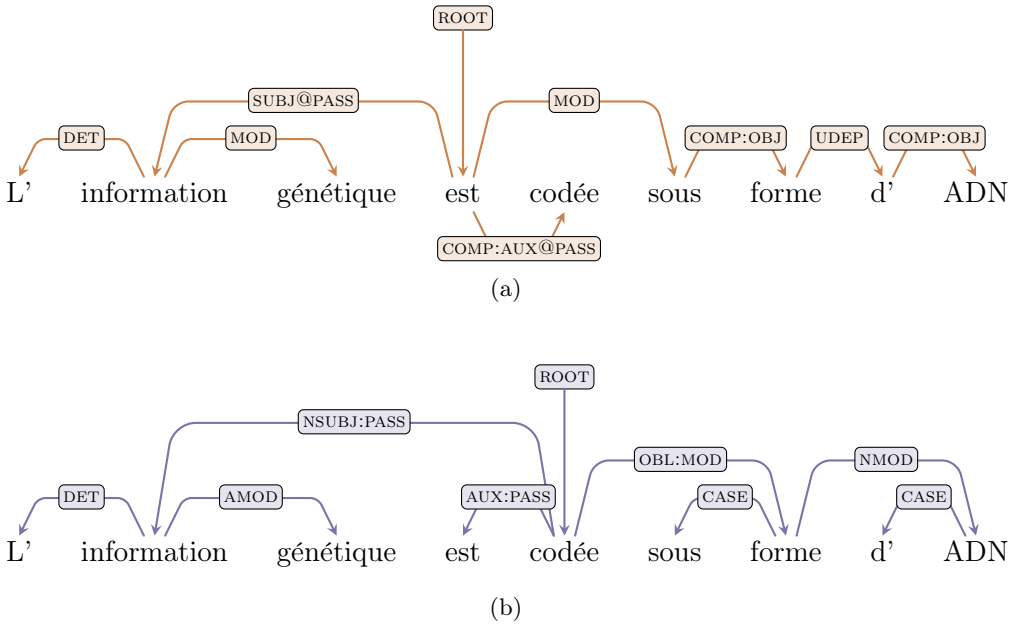


Figure 1.4: Comparison of a SUD tree (a) and a UD tree (b) for the same sentence. Example taken from the French-GSD treebank (Guillaume et al., 2019).

1.4 Parsing Systems

In this section we present the two most common data-driven parsing system types, transition-based and graph-based parsers. The work presented in this thesis exclusively covers supervised learning approaches, so we stick to discussing supervised approaches here despite there existing a plethora of work on semi-supervised and unsupervised parsers.

1.4.1 Transition-based

Transition-based parsers are based on algorithms that predict a given action or transition to take at a given time step based on the current configuration. Transitions are predicted until a terminal configuration is met. A common configuration consists of a stack, buffer, and set of predicted arcs. The stack consists of words that the algorithm has partially processed and

[3] https://surfacesyntacticud.github.io/data/

so at the initial time step it is empty. The buffer contains the words the algorithm has yet to process and at time step zero it contains the full sentence. The set of arcs also starts empty and when the algorithm predicts a transition resulting in an arc, the arc is subsequently added to this set. A standard terminal configuration is when the buffer is empty and only the dummy root is left in the stack. Shown in Figure 1.5 is an example of a run through of the Arc Standard algorithm where there are three transitions, SHIFT, LEFT-ARC, and RIGHT-ARC (Nivre, 2004). The SHIFT transition moves the top of the buffer (b_0) to the stack. The LEFT-ARC transition creates an edge from b_0 to s_0 and pops the top of the stack. The RIGHT-ARC generates an edge from s_0 to b_0 and pops the top of stack and also replaces b_0 with s_0 at the head of the buffer. The full definition of Arc Standard is given in Algorithm 1. As described, this algorithm creates unlabelled trees. The LEFT-ARC and RIGHT-ARC can be labelled (as shown in Example 1.5). So in effect, there would be 75 transitions when working with UD (37 labelled versions of both LEFT-ARC and RIGHT-ARC and the SHIFT transition).

Arc Standard		
Initial configuration:		$c_s(w_0...w_n) = \langle [], [w_0, w_1...w_n], \emptyset \rangle$
Terminal configuration:		$c_t = \langle \Sigma', [], A \rangle$
Transitions:	SHIFT	$\langle \Sigma, b_0 \vert B, A \rangle \Rightarrow \langle \Sigma \vert b_0, B, A \rangle$
	LEFT-ARC	$\langle \Sigma \vert s_0, b_0 \vert B, A \rangle \Rightarrow \langle \Sigma, b_0 \vert B, A(b_0, r, s_0) \rangle$
	RIGHT-ARC	$\langle \Sigma \vert s_0, b_0 \vert B, A \rangle \Rightarrow \langle \Sigma, s_0 \vert B, A(s_0, r, b_0) \rangle$
Preconditions:	SHIFT	*only if* $B_t \neq []$
	LEFT-ARC	*only if* $B_t \neq [] \wedge \Sigma_t \neq [] \wedge s_0 \neq \text{ROOT}$
	RIGHT-ARC	*only if* $B_t \neq [] \wedge \Sigma_t \neq []$

Algorithm 1: Arc Standard. $A(x)$ is shorthand for $A \cup \{x\}$, Σ is the stack, B is the buffer, and A is the set of arcs. And X_t is the initial state of X before applying a transition at time step t. As defined in (Nivre, 2008).

A parser is trained by learning what transition to predict given a particular configuration state:

$$f(\phi_c) = T_c \tag{1.6}$$

where ϕ_c is some feature vector for configuration c that is constructed from a set of features describing the configuration. An example feature set is shown in Example 3. An oracle is used to map configurations to transitions using the gold trees of dataset $\mathbb{D}$ (a set of sentences and their corresponding trees). For each configuration state for each instance in $\mathbb{D}$ this results in a ϕ and a corresponding transition T, such that $\mathbb{D}$ is transformed into a set of feature vectors Φ and the corresponding set of transitions $\mathbb{T}$. The task then is to find the optimal parameters of f such that the set of feature vectors Φ are mapped to a set of predicted transitions $\mathbb{T}'$ such that the difference between the expected transitions $\mathbb{T}$ is minimised.

POS_{s_0}	form_{s_0}	lemma_{s_0}
POS_{s_1}	form_{s_1}	lemma_{s_1}
POS_{b_0}	form_{b_0}	lemma_{b_0}
POS_{b_1}	form_{b_1}	lemma_{b_1}

Example 3: Example of a basic set of features used to construct a feature vector, ϕ, where the POS tag, form, and lemma of the top 2 elements on the stack and buffer are used.

Transition-based algorithms are quite brittle as they are prone to error-propagation, i.e. if the algorithm makes an error it is less likely to make correct predictions at subsequent steps as the configuration is erroneously updated. This can be offset by using a dynamic oracle during

Input: ROOT Whare ghaists and houlets nightly cry

Stack			Buffer				Transition
	s_1	s_0	b_0	b_1	b_2		
			ROOT	Whare	ghaists	...	SHIFT b_0 to top of stack
		ROOT	Whare	ghaists	and	...	SHIFT b_0 to top of stack
	ROOT	Whare	ghaists	and	houlets	...	SHIFT b_0 to top of stack
ROOT	Whare	ghaists	and	houlets	nightly	cry	SHIFT b_0 to top of stack
...	ghaists	and	houlets	nightly	cry		**LEFT-ARC** [cc] remove s_0 from stack
ROOT	Whare	ghaists	houlets	nightly	cry		**RIGHT-ARC** [conj] remove s_0 from stack replace b_0 with s_0
	ROOT	Whare	ghaists	nightly	cry		SHIFT b_0 to top of stack
ROOT	Whare	ghaists	nightly	cry			SHIFT b_0 to top of stack
...	ghaists	nightly	cry				**LEFT-ARC** [advmod] remove s_0 from stack
ROOT	Whare	ghaists	cry				**LEFT-ARC** [nsubj] remove s_0 from stack
	ROOT	Whare	cry				**LEFT-ARC** [mark] remove s_0 from stack
		ROOT	cry				**RIGHT-ARC** [root] remove s_0 from stack replace b_0 with s_0
			ROOT				SHIFT b_0 to top of stack
		ROOT					**Terminal configuration**

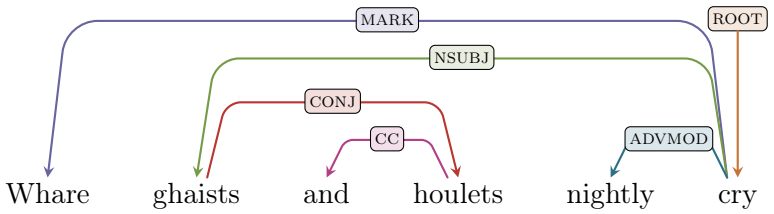


Figure 1.5: An example of an Arc Standard parse for the input text: *Whare ghaists and houlets nightly cry* from Tam o' Shanter (http://www.robertburns.org.uk/Assets/Poems_Songs/tamoshanter.htm) English: *Where ghosts and owls nightly cry.* The resulting dependency tree is also shown.

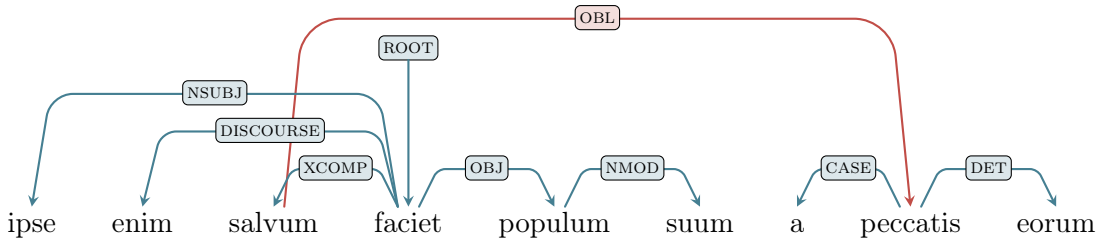


Figure 1.6: An example of non-projective dependency tree (from UD Latin-PROEIL, Mathew 1:21) where the crossing arc is highlighted in red (from *salvum* to *peccatis*). English: *For he shall save his people from their sins.*

training, which updates the expected transitions to best capture the correct dependency tree thus minimising the impact of an error (Gómez-Rodríguez and Fernández-González, 2015). Beyond this, certain transition-based parsers cannot encode certain phenomena that occur in dependency trees. This is most apparent for non-projectivity. A non-projective tree contains at least one edge which crosses another. An example is given in Figure 1.6. Certain algorithms were developed to cater for this, such as Eager Swap and Covington (Nivre, 2009; Covington, 1990, 2001).

Eager Swap allows for non-projectivity by introducing a SWAP transition which takes the top of the stack and places it behind the first element of the buffer, therefore altering the order in which the elements are processed so that nodes that are non-adjacent can be linked by LEFT-ARC or RIGHT-ARC transitions. The Covington algorithm is a list-based algorithm which utilises two lists to store partially-analysed elements. Similar to Arc Standard, arcs are created by LEFT-ARC and RIGHT-ARC transitions but here they are created between the first list Λ_1 and the buffer. It also has a NO-ARC transition which moves the top element in Λ_1 to Λ_2, thus allowing LEFT-ARC and RIGHT-ARC transitions to be applied to non-adjacent elements. The full definitions of these algorithms and their projective counterparts are given in Appendix 1.A. They are used (as well as Arc Standard) for the work described in Section 5.3.

1.4.2 Graph-based

Graph-based parsers fundamentally work by scoring each potential graph for the input by combining scores for parts of the graph. Here we focus on edge-factored scoring of graphs, which score each potential edge of a graph independently. We do this as the graph-based systems used in this thesis are edge-factored parsers. So for a given input, $\mathbf{x}$:

$$f(\mathbf{x}) = \mathbf{W} \tag{1.7}$$

where f is some model and $\mathbf{W} = |\mathbf{x}| \times |\mathbf{x}|$ is a matrix of scores for each potential edge connecting the elements of $\mathbf{x}$. Each element s_{ij} in $\mathbf{W}$ can be considered an estimation of how likely w_i is the head of w_j. Given a dataset $\mathbb{D}$ of sentences $\mathbb{S}$ and their corresponding trees $\mathbb{G}_T$, the task is to obtain the parameters of f such that the edges corresponding to each $\mathcal{G}_\mathcal{T}$ for each sentence $\mathbf{x}$ are higher than the alternative edges. This results in a parser that is natively non-projective, but doesn't ensure well-formed trees are predicted as it can predict graphs that are not fully-connected and that contain cycles. In order to predict a well-formed tree, a maximum (or minimum) spanning tree (MST) algorithm is needed. These find the spanning tree (for dependency trees, technically the spanning arborescence) which has the highest (or lowest) score. Chu-Liu-Edmonds (CLE) is a MST algorithm commonly used in graph-based parsers (Chu and Liu, 1965; Edmonds, 1967). It is the algorithm used by the graph-based systems used throughout this thesis.

Chu-Liu-Edmonds The CLE algorithm is a recursive algorithm consisting of two parts and two functions. A weighted graph $\mathcal{G}$ is passed to **CLE**. The first part of this function

simply predicts a graph $\mathcal{G}'$ based on the maximum score for each node. If this results in a well-formed tree, fully connected and with no cycles, the algorithm ends here and returns this tree. If not, at least one cycle exists in $\mathcal{G}'$. A cycle $\mathcal{C}$ is picked (the order doesn't matter if more than one cycle exists) which is passed to the **contract** function along with $\mathcal{G}'$. At a high level of abstraction, the **contract** function collapses the nodes inside $\mathcal{C}$ and treats them as a single node ν_c. For each outgoing edge ϵ_{ij} for each node ν_i in $\mathcal{C}$, the dependent of the maximum edge is stored in an array ($\text{TRACKER}[\epsilon_{cj}] = \nu_j$). The weight of each incoming edge ϵ_{ij} for each node ν_i outside $\mathcal{C}$ is then updated by subtracting the weight of the predecessor of node ν_j in the cycle (i.e. the edge which would be removed if this incoming edge were added to the graph) and by adding the total score of the edges in $\mathcal{C}$. The function then returns TRACKER and the weighted graph made up of all the nodes except those in $\mathcal{C}$ and with the dummy node ν_c.

This contracted graph is then passed to **CLE** and the contraction process continues until no more cycles are found. At this point the second part of the function **CLE** begins. First, the incoming edge to $\mathcal{C}$ with the maximum score (as saved in $\text{TRACKER}[\epsilon_{ic}]$) is recovered, ν_j. And so the edge ϵ_{ij} is added. Then all the outgoing edges from $\mathcal{C}$ which head the nodes saved in TRACKER are added. All the edges within C are added except the edge ϵ_{kj}, i.e. the edge ending at the node which is headed by the highest edge ϵ_{ij} originating outside $\mathcal{C}$. **CLE** thus returns this expanded graph and iteratively expands in this way until all edges causing a cycle have been replaced and the maximum scoring well-formed tree is returned. An outline of the CLE algorithm is given in Algorithms 2 and 3. Figure 1.7 shows an example of the algorithm in practice for a basic case. Note that this gives an unlabelled tree. The simplest way to obtain a labelled tree is to predict the labels separately or to predict the labels based on the predicted unlabelled tree.

Chu-Liu-Edmonds

Input: $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathbf{W})$
Returns: max spanning tree, $\mathcal{G}_T$

def CLE($\mathcal{G}$):
  Diag($\mathcal{E}$) $= -\infty$ `#no self-headness`
  $\mathcal{E}' = \{\epsilon_{ij}\} | \nu_j \in \mathcal{N}, i = \text{argmax}\, \mathbf{s}_j$ `#get max scoring edges`
  $\mathcal{G}' = (\mathcal{N}, \mathcal{E}', \mathbf{W})$ `#max graph`
  if $\neg \nu_i \rightarrow^* \nu_i \in \mathcal{G}'$ for $\nu_i \in \mathcal{N}$:
    return $\mathcal{G}'$ `#return graph if no cycles`
  Take any $\mathcal{E}_C$ in $\mathcal{E}'$ that contains cycles
  $\mathcal{G}_C$, TRACKER $=$ **contract**$(\mathcal{G}', \mathcal{E}_C)$ `#contract nodes in cycle`
  $\mathcal{N}_T, \mathcal{E}_T, \mathbf{W}_T =$ **CLE**$(\mathcal{G}_C)$ `#pass contracted graph to CLE`
  $\mathcal{G}_T = (\mathcal{N}_T, \mathcal{E}_T, \mathbf{W}_T)$ `#tree with collapsed node`
  $\nu_j = \text{TRACKER}[\epsilon_{ic}]$ `#max outgoing edge from C`
  Take $\nu_k \in \mathcal{N}_C | \epsilon_{kj} \in \mathcal{E}_C$ `#node in C headed by` ν_j
  $\mathcal{E}_T \cup \epsilon_{ij}$ `#add highest edge incoming to C`
  $\mathcal{E}_T \cup \epsilon_{im} \forall \epsilon_{cm} \in \mathcal{E}_C | \nu_i = \text{TRACKER}[\epsilon_{cm}]$ `#max incoming edges to nodes C`
  $\mathcal{E}_T \cup \epsilon_{im} \forall \epsilon_{im} \in \mathcal{E}_C | \epsilon_{im}! = \epsilon_{kj}$ `#add edges in cycle except` ϵ_{kj}
  return $\mathcal{G}_T$

Algorithm 2: Chu-Liu-Edmonds. **Contract** is in Algorithm 3

Contract

Input: $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathbf{W})$ and $\mathcal{C}$
Returns: contracted graph $\mathcal{G}_c$ and TRACKER that saves max edges to/fro cycle

def contract$(\mathcal{G}, \mathcal{C})$:
  $\mathcal{N}_C = \mathcal{N} / \mathcal{C}$ `#set of nodes excluding C`
  $\mathcal{E}_C = \mathcal{E} / \mathcal{C}$ `#set of edges excluding C`
  $\mathcal{N}_C \cup \{\nu_c\}$ `#collapse cycle to node and add to graph`
  $\mathbf{W}_C = \mathbf{W}(r_i; c_i)\ \forall \nu_i \in \mathcal{C}$ `#remove rows and colums for nodes in C`
  $\mathbf{W}_C = \begin{bmatrix} \mathbf{W}_C \\ \mathbf{0} \end{bmatrix}$, $\mathbf{W}_C = \begin{bmatrix} \mathbf{W}_C & \mathbf{0} \end{bmatrix}$ `#add row and column for new node`
  For $\nu_j \in \mathcal{E}_C \mid \exists \nu_i \in \mathcal{C}\ \exists \epsilon_{ij} \in \mathcal{E}$: `#evaluate outgoing edges from C`
    $\mathcal{E}_C \cup \{\epsilon_{cj}\}$ `#add outgoing edges from C`
    $\nu_j = \text{argmax}\, \mathbf{s}_j \mid \epsilon_{ij} \in \mathcal{C}$ `#get highest outgoing`
    TRACKER$[\epsilon_{cj}] = \nu_i$ `#track highest outgoing`
    $\mathbf{W}_{C_{cj}} = \mathbf{W}_{ij}$ `#set weight of edge for contracted graph`
  For $\nu_i \in \mathcal{E}_C \mid \exists \nu_j \in \mathcal{C}\ \exists \epsilon_{ij} \in \mathcal{E}$ `#evaluate edges incoming to C`
    $\mathcal{E}_C \cup \{\epsilon_{ic}\}$ `#add incoming edges to C`
    $\nu_j = \text{argmax}\, \mathbf{s}_j - \mathbf{s}_{a(j)} \mid \epsilon_{ij} \in \mathcal{C}$ `#get highest incoming`
    TRACKER$[\epsilon_{ic}] = \nu_j$ `#track highest incoming`
    $\mathbf{W}_{C_{ic}} = \mathbf{W}_{ij} - \mathbf{W}_{a(j)j} + \text{score}(\mathcal{C})$ `#set weight of edge for contracted graph`
  $\mathcal{G}_C = (\mathcal{N}_C, \mathcal{E}_C, \mathbf{W}_C)$
  return $\mathcal{G}_C$, TRACKER

where $a(j)$ is the predecessor of ν_j in $\mathcal{C}$
and $\text{score}(\mathcal{C}) = \sum_{\epsilon_{ij} \in \mathcal{C}} \mathbf{W}_{ij}$

Algorithm 3: Contract function of Chu-Liu-Edmonds.

1.4.3 EVALUATION

The unlabelled attachment score (UAS) is the F1 score of the head position of each token:

$$\text{UAS} = \frac{2 \times \text{correct}}{\text{predicted} + \text{gold}} \tag{1.8}$$

where correct is the number of correct predictions, predicted is the total number of edges predicted, and gold is the number of edges in the gold-labelled data. For gold-tokenized data this simplifies to just the accuracy (correct/gold). UAS for gold-tokenized data is basically the edit distance between the gold and predicted trees divided by the number of nodes. As in the distance between the gold tree and the predicted tree is measured in the number of edits required to morph the predicted tree to the gold one. So for each erroneous edge the distance increases by one. The labelled attachment score (LAS) is the same except a correct head prediction with an incorrect label is considered one edit. So it is feasible that the evaluation of dependency parsers could use one of the many measurements between graphs that penalise differences more severely (Wills and Meyer, 2020). Linguistically-grounded measures of distance could also be used (Kummerfeld et al., 2012). There already exist metrics which take other syntactic related information in consideration such as morphological analysis (Zeman et al., 2018). However, we use UAS and LAS throughout and sometimes forsake the unlabelled measurement for the more syntactically informative labelled version when analysis becomes too crowded by reporting both.

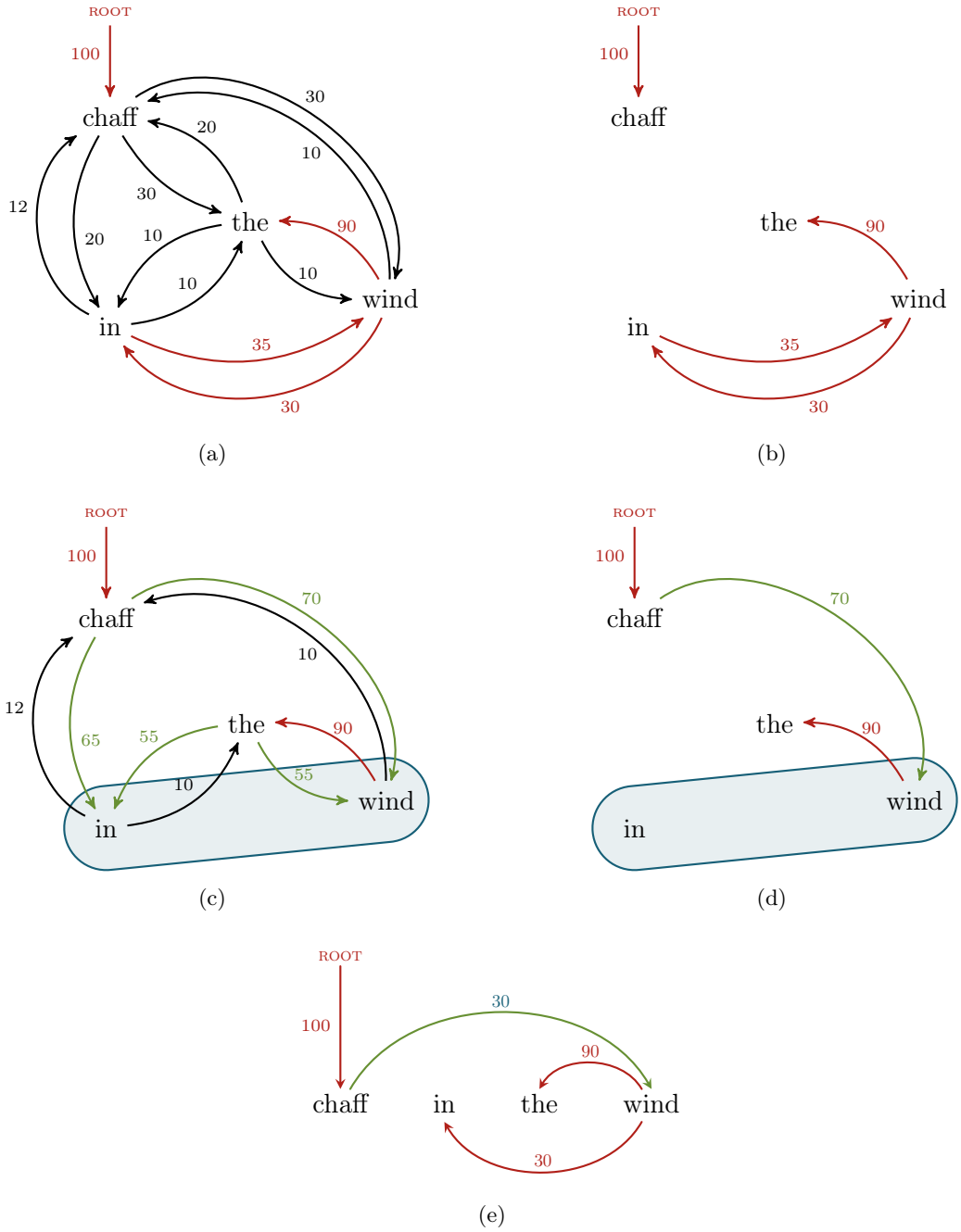


Figure 1.7: An example of the Chu-Liu-Edmonds algorithm to obtain the MST. Only the score for edges from the dummy root node are shown for *chaff* for the sake of space: consider the scores for the other nodes to be infinitesimally small and can be ignored. Based on the scores in (a), the maximum scores result in the graph shown in (b). There is a cycle between *in* and *wind*, so these nodes are contracted into a placeholder node. The scores of incoming edges to this placeholder are updated by taking away the score of the edge coming from the predecessors of the nodes in the cycle (the score effectively disallowed) and adding the total score of the edges in the cycle, e.g. the edge *chaff*→*wind* is $30-35+75$ as the original incoming edge is 30, the edge to *wind* from its predecessor in the cycle *in* is 35, and the total score of the edges in the cycle is 75. These edges are shown in (c). Based on these new edge scores, the maximum edges associated with the graph containing the placeholder node are used as shown in (d). This graph contains no cycles, so the placeholder node is expanded by taking the highest allowed edge in the cycle (*wind*→*in*) and the final parse results in the dependency tree shown in (e).

1.4.4 A note on neural networks

While any classifier can be used with the parsing systems mentioned above and those used throughout this thesis, the majority of work in this thesis was undertaken using neural networks. It doesn't seem particularly useful nor interesting to describe neural networks in any great detail here. However, we give a very brief description of bidirectional long-term short-term networks and networks in general. Fundamentally, tokens are represented in some continuous fashion so they take the form of a vector. These continuous representations or embeddings are then fed to a neural network. The simplest network, a Perceptron, is a linear classifier which uses linear combinations of weights and the input to predict a given class (Rosenblatt, 1958).

BiLSTMs Long-short short-term (LSTM) networks are a type of recurrent neural network (RNN) (Hochreiter and Schmidhuber, 1997). They offset the tendency for RNNs to suffer from vanishing gradients by using a cell which remembers varying degrees of information over arbitrary time steps by using a forget gate. The LSTM cell consists of an input, output and forget gate. The input gate decides to what degree to add new information and the output gate decides how much information encoded in the cell is sent to the network at the subsequent time step. The forget gate allows the LSTM to decide what information should or should not be forgotten. The gates have activation functions (e.g. sigmoid, rectified linear, hyperbolic tangent) which regulate the flow of information that comes in and out of the cell (or that is forgotten). Then based on the state of the cell(s) at a given time step a hidden representation is obtained for the input of the network (in our case some representation of a token). In this thesis, most neural parsers used rely on bidirectional LSTM (BiLSTM) networks which connect the output of two hidden layers which process the data in opposite directions into one output (Schuster and Paliwal, 1997). BiLSTMs are particularly useful for NLP as they learn to encode the pertinent information about the context a particular token or instance occurs in.

1.5 Summary

The details presented in this chapter should be sufficient information to understand each of the subsequent chapters in isolation for most readers interested in this thesis. The description of dependency grammar and the UD framework is enough to follow the work and where more information is required for a specific piece of work, we describe that in the respective chapters. The details on the parsing systems give a broad idea of dependency parsing in NLP. This will be expanded on in the following chapter but with less formal descriptions as we discuss the recent work in this field.

1.A TRANSITION-BASED ALGORITHMS

ARC EAGER		
Initial configuration:		$c_s(w_0...w_n) = \langle [], [w_0, w_1...w_n], \emptyset \rangle$
Terminal configuration:		$c_t = \langle [w_0], [], A \rangle$
Transitions:	SHIFT	$\langle \Sigma, b_0 \vert B, A \rangle \Rightarrow \langle \Sigma \vert b_0, B, A \rangle$
	REDUCE	$\langle \Sigma \vert s_0, B, A \rangle \Rightarrow \langle \Sigma, B, A \rangle$
	LEFT-ARC	$\langle \Sigma \vert s_0, b_0 \vert B, A \rangle \Rightarrow \langle \Sigma, b_0 \vert B, A(b_0, r, s_0) \rangle$
	RIGHT-ARC	$\langle \Sigma \vert s_0, b_0 \vert B, A \rangle \Rightarrow \langle \Sigma \vert s_0, b_0, B, A(s_0, r, b_0) \rangle$
Preconditions:	SHIFT	*only if* $B_t \neq []$
	REDUCE	*only if* $\Sigma_t \neq [] \wedge (w_i, r^*, s_0) \in A_t$
	LEFT-ARC	*only if* $B_t \neq [] \wedge \Sigma_t \neq []$ $\wedge\, s_0 \neq \text{ROOT} \wedge (w_i, r^*, s_0) \notin A_t$
	RIGHT-ARC	*only if* $B_t \neq [] \wedge \Sigma_t \neq []$

Algorithm 4: Arc Eager. $A(x)$ is shorthand for $A \cup \{x\}$, Σ is the stack, B is the buffer, A is the set of arcs, and X_t is the initial state of X before applying a transition at time step t (Nivre, 2008).

SWAP EAGER		
Initial configuration:		$c_s(w_0...w_n) = \langle [], [w_0, w_1...w_n], \emptyset \rangle$
Terminal configuration:		$c_t = \langle \Sigma', [], A \rangle$
Transitions:	SHIFT	$\langle \Sigma, b_0 \vert B, A \rangle \Rightarrow \langle \Sigma \vert b_0, B, A \rangle$
	LEFT-ARC	$\langle \Sigma \vert s_0, b_0 \vert B, A \rangle \Rightarrow \langle \Sigma, b_0 \vert B, A(b_0, r, s_0) \rangle$
	RIGHT-ARC	$\langle \Sigma \vert s_0, b_0 \vert B, A \rangle \Rightarrow \langle \Sigma, s_0 \vert B, A(s_0, r, b_0) \rangle$
	SWAP	$\langle \Sigma \vert s_0, b_0 \vert B, A \rangle \Rightarrow \langle \Sigma, b_0 \vert s_0 \vert B, A \rangle$
Preconditions:	SHIFT	*only if* $B_t \neq []$
	LEFT-ARC	*only if* $B_t \neq [] \wedge \Sigma_t \neq [] \wedge s_0 \neq \text{ROOT}$
	RIGHT-ARC	*only if* $B_t \neq [] \wedge \Sigma_t \neq []$
	SWAP	*only if* $B_t \neq [] \wedge \Sigma_t \neq []$ $\wedge\, x_{b_0} > x_{s_0} > 0$

Algorithm 5: Swap Eager. $A(x)$ is shorthand for $A \cup \{x\}$, Σ is the stack, B is the buffer, A is the set of arcs, and X_t is the initial state of X before applying a transition at time step t (Nivre, 2009).

COVINGTON PROJECTIVE		
Initial configuration:		$c_s(w_0...w_n) = \langle [], [], [w_0, w_1...w_n], \emptyset \rangle$
Terminal configuration:		$c_t = \langle \Lambda'_1, [], [], A \rangle$
Transitions:	SHIFT	$\langle \Lambda_1, \Lambda_2, b_0 \vert B, A \rangle \Rightarrow \langle \Lambda_1 \frown \Lambda_2 \vert b_0, [], B, A \rangle$
	LEFT-ARC	$\langle \Lambda_1 \vert \lambda_0, \Lambda_2, b_0 \vert B, A \rangle \Rightarrow \langle \Lambda_1, [], b_0 \vert B, A(b_0, r, \lambda_0) \rangle$
	RIGHT-ARC	$\langle \Lambda_1 \vert \lambda_0, \Lambda_2, b_0 \vert B, A \rangle \Rightarrow \langle \Lambda_1 \vert \lambda_0 \vert b_0, [], B, A(\lambda_0, r, b_0) \rangle$
	NO-ARC	$\langle \Lambda_1 \vert \lambda_0, \Lambda_2, B, A \rangle \Rightarrow \langle \Lambda_1, \lambda_0 \vert \Lambda_2, B, A \rangle$
Preconditions:	SHIFT	*only if* $B_t \neq []$
	LEFT-ARC	*only if* $B_t \neq [] \wedge \Lambda_{1_t} \neq [] \wedge \lambda_0 \neq \text{ROOT}$
	RIGHT-ARC	*only if* $B_t \neq [] \wedge \Lambda_{1_t} \neq []$
	NO-ARC	*only if* $\Lambda_{1_t} \neq [] \wedge \ (w_i, r^*, \lambda_0) \in A_t$

Algorithm 6: Covington projective. Where Λ_i is a list; $X \frown Y$ means the concatenation of the lists X and Y; B is the buffer, A is the set of arcs, and X_t is the initial state of X before applying a transition at time step t (Covington, 2001; Nivre, 2008).

COVINGTON NON-PROJECTIVE		
Initial configuration:		$c_s(w_0...w_n) = \langle [], [], [w_0, w_1...w_n], \emptyset \rangle$
Terminal configuration:		$c_t = \langle \Lambda'_1, [], [], A \rangle$
Transitions:	SHIFT	$\langle \Lambda_1, \Lambda_2, b_0 \vert B, A \rangle \Rightarrow \langle \Lambda_1 \frown \Lambda_2 \vert b_0, [], B, A \rangle$
	LEFT-ARC	$\langle \Lambda_1 \vert \lambda_0, \Lambda_2, b_0 \vert B, A \rangle \Rightarrow \langle \Lambda_1, \lambda_0 \vert \Lambda_2, b_0 \vert B, A(b_0, r, \lambda_0) \rangle$
	RIGHT-ARC	$\langle \Lambda_1 \vert \lambda_0, \Lambda_2, b_0 \vert B, A \rangle \Rightarrow \langle \Lambda_1, \lambda_0 \vert \Lambda_2, b_0 \vert B, A(\lambda_0, r, b_0) \rangle$
	NO-ARC	$\langle \Lambda_1 \vert \lambda_0, \Lambda_2, B, A \rangle \Rightarrow \langle \Lambda_1, \lambda_0 \vert \Lambda_2, B, A \rangle$
Preconditions:	SHIFT	*only if* $B_t \neq []$
	LEFT-ARC	*only if* $B_t \neq [] \wedge \Lambda_{1_t} \neq [] \wedge \lambda_0 \neq \text{ROOT}$ $\wedge \ (w_i, r^*, \lambda_0) \notin A_t \wedge \ \lambda_0 \rightarrow^* b_0 \notin A_t$
	RIGHT-ARC	*only if* $B_t \neq [] \wedge \Lambda_{1_t} \neq []$ $\wedge \ (w_i, r^*, b_0) \notin A_t \wedge \ b_0 \rightarrow^* \lambda_0 \notin A_t$
	NO-ARC	*only if* $\Lambda_{1_t} \neq []$

Algorithm 7: Covington non-projective. Where Λ_i is a list. $X \frown Y$ means the concatenation of the lists, X and Y (Covington, 2001; Nivre, 2008).

Chapter 2

Recent Work

In this chapter, we give a snapshot of dependency parsing in the recent past. In Section 2.1, we give an overview of recent work relating to dependency parsing, dependency grammar, and syntax in NLP. Then in Section 2.2, we discuss the current state of dependency parsers pertaining to accuracy and inference speed. This chapter is by no means exhaustive but should serve as a decent setting for understanding how the work presented in this thesis fits into the wider landscape.

2.1 Parsing and syntax in the recent past

Currently NLP parsing is not relatively popular. Figure 2.1 shows the percentage and absolute number of papers submitted to the Tagging, Chunking, Syntax and Parsing track at ACL conferences from 2013 to 2020. A clear downward trend can be seen after 2015 for the percentage of submissions, resulting in only about 2% of the total submissions. Note too that this track isn't exclusively for parsing. Parsing has become a niche corner of NLP. However, it should be noted that this holds for the relative popularity of syntactic work at ACL (one outlet for NLP work) and that the absolute number of submissions to this track doesn't follow the same downward trend. Of course, the popularity of a field is neither here nor there, but it is indicative of the state of NLP in this moment of time.

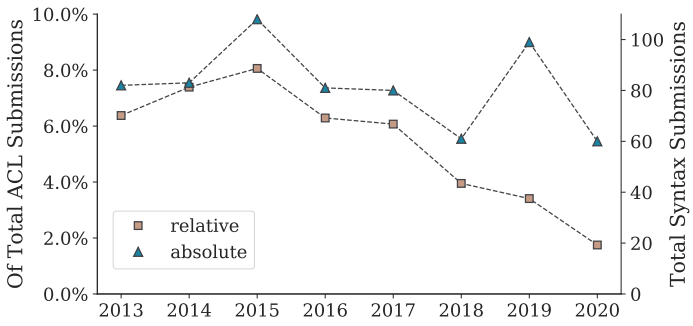


Figure 2.1: The waning popularity of the Tagging, Chunking, Syntax and Parsing track at ACL conferences. Numbers taken from reports at https://www.aclweb.org/adminwiki.

2.1.1 Non-neural parsing

In Chapter 1, we exclusively discussed data-driven dependency parsing. This isn't the only approach. In the past, you could still see people using grammar-driven techniques (Hays, 1964; Gaifman, 1965; Lombardo and Lesmo, 1996; Tapanainen and Jarvinen, 1997; Järvinen

and Tapanainen, 1998; Eisner, 2000). In fact, fairly recently there is an example of a study comparing data-driven and grammar-driven techniques for dependency parsing (Haverinen et al., 2009). Even more recently, Gamallo (2015) introduced a purely symbolic grammar-driven parser.[1] Grammar-driven techniques use a grammar which consists of a set of rules to apply to linguistic data. These methods are very brittle. First, it is likely impossible to develop an exhaustive grammar and so it is likely that the parsers will come across sentences that don't fit their grammar. Second, these formal grammars can't process ungrammatical input. But they are more linguistically grounded and much more interpretable.

Latterly, data-driven parsers became more popular. These typically used one of the two methods presented in the preceding chapter. Some transition-based systems used instance-based classifiers (Nivre et al., 2004) or support vector machines such as MaltParser, which we use in Section 5.3 because it has numerous algorithms implemented (Nivre et al., 2007). Transition-based parsers using perceptrons as their classifiers were also popular, for example Huang and Sagae (2010) did so with dynamic programming, Zhang and Nivre (2011) with non-local features, Bohnet and Nivre (2012) with joint training of POS tags. Others used perceptrons for graph-based parsers (Carreras et al., 2006) and linear classifiers (McDonald et al., 2005; McDonald and Pereira, 2006). Before the deep learning storm approached, ensemble systems became popular where multiple parsers were trained and their predictions were combined resulting in more rigorous parsers (Sagae and Tsujii, 2010; Surdeanu and Manning, 2010; Lavelli, 2012; Green et al., 2012).

2.1.2 Neural dependency parsing

Neural dependency parsing didn't start off with a bang. Titov and Henderson (2007) introduced what was retrospectively named the first neural dependency parser, which was on par with current systems. Later, Stenetorp (2013) used a vanilla RNN with a transition-based system. They obtained a parser that was competitive with current systems without hand-crafted features, but still performed significantly worse, e.g. 6 UAS points less than the ensemble parser of Surdeanu and Manning (2010) on the Penn Treebank (PTB) (Marcus and Marcinkiewicz, 1993). Similarly, Socher et al. (2010) had used RNNs for constituency parsing and obtained a similar result: a parser without extensive manually devised features approaching the performance of the best parsers of the day.

Then Chen and Manning (2014) introduced a simple feed forward network for dependency parsing. With it, they used a transition-based system using the Arc Standard algorithm. The input layer was made up of a concatenation of lexical, POS tag, and relation label features. Each of these were based on the current configuration and were themselves a concatenation of representations (i.e. the words at the top of the stack and buffer). They used a fairly light feature vector compared to classical systems, but it still contained 25 features. This input was then fed to a single linear layer, but they used their own *cube* activation function introduced in the same paper. The last layer was a softmax for computing the probability for a given transition for each time step given the current configuration. They obtained improvements over MSTParser (the graph-based parser of McDonald et al. (2005)) and MaltParser with respect to accuracy and speed. Weiss et al. (2015) introduced a very similar model adding an extra perceptron to have a beam-based system. They also utilised tri-training which is a version of self-training where two parsers are used to label new data for training but only if the two parsers agree exactly for a given instance, thus allowing them to train with more data. They obtained significant improvement over the original feed forward parser.

Dyer et al. (2015) extended the use of neural networks by using LSTMs. They developed a system where the stack, the buffer, and transition history all have a separate LSTM layer. They didn't use a feature vector, but instead used the current LSTM output state for each LSTM such that transitions were predicted based on the current representation of the stack,

[1] There are even some stalwarts holding out in the neural era (Chiruzzo and Wonsever, 2020).

buffer, and transition history. Like Chen and Manning (2014), they used Arc Standard for the algorithm and it outperformed their system on both PTB and the Chinese Penn Treebank (CTB) (Xue et al., 2005). However, it didn't outperform the system from Weiss et al. (2015).

Kiperwasser and Goldberg (2016) further developed neural dependency parsing by implementing both a transition-based and graph-based parser using a BiLSTM network. For the transition-based system, they concatenated the hidden representations from the BiLSTM layers for certain tokens depending on the configuration at a given time step. They found that they only needed a very small amount of context which entailed using the hidden representations of the top three tokens on the stack and the top item on the buffer. They speculate that extra context isn't needed as the BiLSTM layers already encode sufficient context. They used Arc Hybrid for the algorithm. The Arc Hybrid algorithm is a combination of Arc Standard and Arc Eager (Kuhlmann et al., 2011). It takes the RIGHT-ARC transition from the former (except the transition RIGHT-ARC creates an edge from s_1 to s_0 on the stack and pops s_0 from the stack in contrast to the definition in Algorithm 1) and takes SHIFT and LEFT-ARC from Arc Eager. The graph-based system used the arc-factored approach of McDonald et al. (2005) where they score each edge independently and then use Eisner's algorithm to find the highest scoring projective tree (Eisner, 1996). They used a simple MLP as the scoring function to which only the concatenated output from the BiLSTM for the token and a potential head were fed (this for all potential pairs). Using these systems with minimal features, the transition-based system was on par with the leading system at the time (still (Weiss et al., 2015)) although the best performing system from Kiperwasser and Goldberg (2016) was a transition-based parser with 11 features. Their graph-based system lagged behind a little, but was a much lighter network and had much fewer manually configured components. In contrast, Kuncoro et al. (2016) used an ensemble of parsers and then distilled them into one graph-based parser. The parser was very accurate, but also fairly slow.

Zhang et al. (2014) had shown the usefulness of utilising characters by using character features for their parser and obtained increases in parsing performance for CTB. Ballesteros et al. (2015) developed the parser of Dyer et al. (2015) by utilising character embeddings (Dos Santos and Zadrozny, 2014; Zhang and LeCun, 2015). They obtained considerable improvements over the original parser, especially for agglutinative languages. They also compared their system to parsers which utilised careful handling of morphological features and obtained similar results, effectively showing character embeddings captured at least some sort of morphological information. Besides the merits of character embeddings with respect to morphology, they also temper out-of-vocabulary issues.

Biaffine Zhang et al. (2017) introduced an arc-factored graph-based neural network parser that was very competitive despite being conceptually simple. The parser was trained for head selection, meaning for each token a head was selected based on the probability distribution over tokens in a given instance. Dozat and Manning (2018) introduced a similar graph-based neural network that also predicted the head of each token. The main contribution was the biaffine attention mechanism. Two MLPs are separately applied to the output of the last BiLSTM layer. This reduces the dimensions and is hypothesised to create representations for each token as a dependent and as a head. The output of both of these MLPs is then passed to the biaffine attention layer where they undergo two affine transformations, first the transpose of the output from the head MLP is applied to a set of weights and a bias term and then the weights are applied to the output of the dependent MLP. A standard affine transformation would miss out the first transformation. This results in a matrix of scores with $b \times N \times N$ dimensions where b is the number of instances in a given batch and N is the length of the instances (the max length with padding elsewhere). The biaffine mechanism is

then used to obtain the score of a given edge s_{ij} from w_i to w_j:

$$s_{ij} = \mathbf{h}_i^\top \mathbf{W}\mathbf{d}_j + \mathbf{h}_i^\top \mathbf{b} \tag{2.1}$$

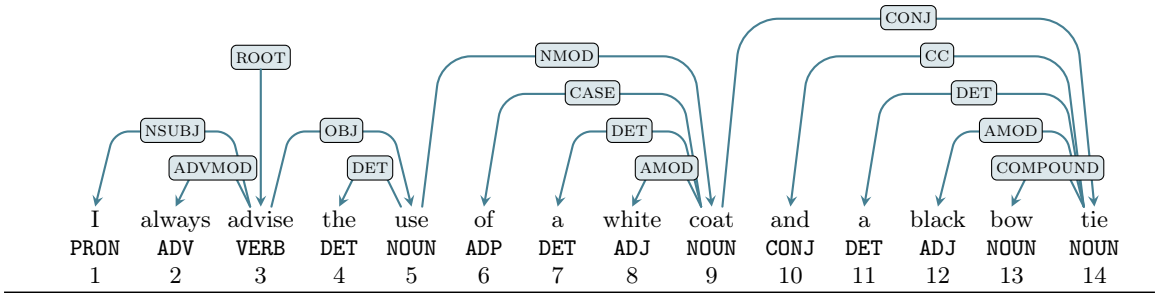
where h_i is the output from the MLP layer encoding the hidden representations as head for token w_i, d_j is the output from the MLP layer encoding the hidden representations of token w_j as a dependent, $\mathbf{W}$ is the weights of the biaffine layer, and b is the bias. Similarly, a biaffine mechanism is used to predict the labels, but the dimensions of $\mathbf{W}$ are such that the resulting dimensions are $b \times N \times N \times R$ where R is the size of the label space and two separate MLPs are used to encode the hidden representation for the label biaffine mechanism. The CLE algorithm (Algorithms 2 and 3 in Section 1.4.2) is used on the edge scores from Equation 2.1 to enforce a well-formed tree.

The initial implementation was competitive with the leading system at the time (the ensemble of parsers from Kuncoro et al. (2016)) using pre-trained embeddings and POS tag embeddings. The authors mainly focused on comparing their parser with other systems using PTB and CTB. They did look at a small sample of other treebanks from the CoNLL 09 shared task. The same system was augmented with character embeddings at the CoNLL 2017 shared task where it clearly outperformed the other systems on the 81 UD treebanks (Dozat et al., 2017; Zeman et al., 2017). The second best system achieved an average LAS 1.3 points lower than the biaffine parser despite being a weighty ensemble system combining two transition-based parsers and a graph-based one (Shi et al., 2017).

Pointer Networks Pointer networks use an attention mechanism that selects one input element to *point* to at a given time step and uses previous selections to influence subsequent ones (Vinyals et al., 2015). Ma et al. (2018) extended the pointer network to include a stack and used a biaffine attention mechanism to point to a word and select the corresponding *transition*. It has two transitions: ARC and REDUCE. ARC is predicted if the attention mechanism points to a word w_i where the word at the top of the stack is w_j and $i \neq j$. If ARC is predicted, then an edge from w_j to w_i is predicted and w_i is pushed to the top of the stack. If the attention mechanism points to the word on top of the stack, REDUCE is predicted. REDUCE pops the word at the top of the stack. This results in a top-down approach with $2n - 1$ steps per input of length n. They obtained leading performance on the treebanks they tested their system on, and at the time their system was released, it was one of the best with respect to PTB. We refer to this parser as Pointer-TD in Table 2.1.

Fernández-González and Gómez-Rodríguez (2019) also used pointer networks but simplified the top-down approach of Ma et al. (2018) by using a left-to-right parser (referred to as Pointer-LR in Table 2.1). Their approach does not require a stack and halves the number of steps needed for a parse (n transitions for input of length n). The system processes the input sequentially starting at the first token and the attention mechanism points to the head of the current word. In this way, previous predictions affect subsequent ones (in contrast to the graph-based biaffine where the predictions are independent). It is conceptually related to transition-based algorithms with non-local transitions, i.e. transitions that can create edges between non-adjacent tokens by selecting the *nth* element on the stack (Fernández-González and Gómez-Rodríguez, 2018).

Sequence-labelling Recently, efforts have been made to simplify the process of parsing sentences by casting the task as sequence labelling. This entails encoding the tree structures in such a way that each token in a sentence has a tag that needs to be predicted. Figure 2.2 shows a full parse tree along with the encodings described here. Spoustová and Spousta (2010) encoded dependency trees as an individual label for each token consisting of the head's POS tag, the *distance* and direction the head is from the token, and the dependency relation. The distance is not actually a distance but the position of the head compared to other tags of the same type. So for the token *a* in Figure 2.2 at position 11 that is headed



abs	3	3	0	5	3	9	9	9	5	14	14	14	14	9
rel1	R2	R1	L3	R1	L2	R3	R2	R1	L4	R4	R3	R2	R1	L5
rel2	V-R1	V-R1	R-L1	N-R1	V-L1	N-R1	N-R1	N-R1	N-L1	N-R2	N-R2	N-R2	N-R1	N-L2
bra	_	<	<\\	/	<>	/	<	<	<\\\>	/	<	<	<	<\\\\>

Figure 2.2: Example of different sequence-labelling encodings. V is short for `VERB`, R is short for `ROOT`, and N is short for `NOUN`. abs is the absolute positional encoding used by Strzyz et al. (2019b) as a baseline encoding; rel1 is the relative positional encoding of Li et al. (2018b); rel2 is the relative positional encoding of Spoustová and Spousta (2010); and bra is the bracketing encoding introduced by Strzyz et al. (2019b). We don't append the dependency relations here for the sake of space. Note that for *advise* the distance and direction to its head is encoded based on the dummy root note being at position 0. Text is an extract from de Fleury (1934).

by *tie* at position 14, the label would be **NOUN-2R-det** as another noun *bow* precedes *tie*. They only reported an opaque result that systems trained using this encoding were "*5-10% below state-of-the-art*", presumably on the Czech and English data from CoNLL 2009 as that is what was used for evaluating the coverage of the encoding (Hajič et al., 2009). Li et al. (2018b) introduced a slightly different encoding scheme. For each token a label is predicted consisting of `L` (when the head is left of the token) or `R` (when the head is right) followed by the distance ($d = |x_i - x_j|$ where x_n is the position of w_n in the sentence and w_i is the head of token w_j). So the for *a* at position 11 in Figure 2.2, the label would be **R3-det**.

Gómez-Rodríguez and Vilares (2018) introduced an ancestral encoding for constituency parsing which performed admirably with respect to accuracy when compared to leading systems and was considerably faster, whereas previous attempts had proven to much less viable. Strzyz et al. (2019b) then compared the relative positional encodings of Spoustová and Spousta (2010) and Li et al. (2018b) against a simple absolute positional encoding (i.e. the index of the head for a given token) and a bracketing encoding based on the work of Yli-Jyrä and Gómez-Rodríguez (2017). The bracketing encoding consists of a series of elements for the label of w_i: $<$ if w_{i-1} has an incoming edge from the left; $n \setminus$ where n is the number of outgoing edges from w_i going to the left; $n /$ where n is the number of outgoing edges from w_{i-1}; $>$ if w_i has an edge incoming from the left. And then the dependency label is concatenated with this bracketing encoding. So for *a* at position 11 in Figure 2.2, the label would contain **det** and $<$: *and* (w_{i-1}) has one incoming left edge, so we include $<$; *a* has no outgoing edges to the left, so no \, *and* has no outgoing edges to the right, so no /; and *a* has no incoming edge from the right. This encoding can only deal with fully projective trees and so recently a bracketing encoding for 2-planar trees has been developed which covers 99.9% of the cases of non-projectivity for the data analysed (Strzyz et al., 2020). Strzyz et al. (2019b) obtained results that showed sequence-labelling parsing was a viable alternative and was especially useful when considering parsing speed.

Vilares et al. (2019) improved the system of Gómez-Rodríguez and Vilares (2018) in a number of ways, but the clearest improvements comes from casting the sequence-labelling task into a multi-task learning (MTL) setup where each component of the label is considered a separate tag. This not only improved accuracy, it also improved the speed of the parser. Strzyz et al. (2019a) confirmed the efficacy of this approach for sequence-labelling dependency

parsing where the dependency label was predicted as another task. They also found improvements when using a different formalism as an auxiliary task in a MTL setup (i.e. using constituency parsing as a task in the MTL setup which contributes less to the loss). Beyond this, Gómez-Rodríguez et al. (2020) demonstrated the theoretical relation between transition-based parsing and sequence-labelling parsing, meaning new encodings can easily be obtained.

2.1.3 Life under the Biaffine hegemony

In the CoNLL 2018 Shared Task on Multilingual Parsing from Raw Text to Universal Dependencies, the top performing models were all based on a biaffine parser (Zeman et al., 2018). The top performing model was the original biaffine parser with contextualised word embeddings from Peters et al. (2018) and using an ensemble of 3 parsers (Che et al., 2018). Kanerva et al. (2018) came second with an implementation of the original biaffine system despite focusing their efforts on developing a neural machine translation based lemmatiser. The highest ranked transition-based parser was UUParser out of Uppsala (Smith et al., 2018). It came joint 7th (tied with Qi et al. (2018), the contribution from Stanford which too was an extension of the original biaffine). Their submission was truly multilingual with models trained on multiple languages such that they only had 34 models spanning 82 treebanks (although note that some languages have more than one treebank). Of the 24 submissions that submitted system descriptions, 14 systems were graph-based of which 13 were extensions of the biaffine parser. The remaining 10 were transition-based with one using a biaffine mechanism for their transition classifier (the stack pointer network parser of Li et al. (2018c)). The baseline model was UDPipe 1.2 (which some submissions also used), also a transition-based parser (Straka et al., 2016).

Nguyen and Verspoor (2018), the only graph-based system not to use the biaffine mechanism, extended the graph-based BIST parser to incorporate tagging. They implement a soft sharing MTL network where the tagger had its own BiLSTM layers and the predicted tags from the tagger were fed to the parser. While they improved the performance of the original implementation (achieving LAS of 92.87 compared to the original of 91.9 on PTB) and achieved one of the leading POS tag performances on PTB (97.97), their system came 14th at the CoNLL 2018 shared task (out of 26 systems). This highlights the poverty of using a single treebank on a single language on a single domain for evaluating which systems are the best. This system is less than a point worse than the biaffine parse on PTB, but achieves almost 4 points less on the average performance on the UD treebanks when compared to the system most similar to it except it uses a biaffine mechanism (Qi et al., 2018).

Straka (2018) obtained speeds (for both tagging and parsing) of up to 624 tokens/second on CPU (16 threads) and 2790 tokens/second (4 threads) using a GeForce GTX 1080 with Tensorflow 1.5.1 while coming joint third at the CoNLL 2018 Shared Task with an average LAS of 73.11. Their model is a multi-task BiLSTM system which jointly predicts UPOS tags, lemmas, and dependencies (using a biaffine mechanism). They tested both a loosely joint model and a tightly joint model. In the former, only the word embeddings are shared and the parsing system is given the predicted UPOS tags as input (predicted are used both at training and inference time). In the tightly joint model, the BiLSTM layers are shared in training (and so the parser only has indirect input from the UPOS tags). They found that the tightly joint model performed best with respect to dependency parsing, but the tagger was slightly less accurate and the lemmatizer considerably less so when compared to the loosely joint model.

Similarly at the EUD Shared Task 2020 at IWPT (Bouma et al., 2020), 5 systems (including our own) used the biaffine parser as the main underlying model of their contribution, 2 used the semantic parser (capable of predicting graph structures rather than just trees) based on the biaffine parser (Dozat and Manning, 2018) and 1 contribution used their own semantic parser based on the biaffine model (Wang et al., 2019, 2020). The remaining 2 systems were

transition-based parsers.

Although the pointer network transition based systems are the leading systems on PTB (and achieve high scores on a narrow subset of UD treebanks), when compared to other systems in a much more challenging setting they haven't fared so well. For example, Li et al. (2018c) submitted a model that jointly predicted UPOS and parsing with the stack pointer network of Ma et al. (2018) to the CoNLL 2018 shared task. It was the third best transition-based system at the shared task.

UUParser is the current leading *classical* transition-based parser. It is an extension of BIST, using the Arc Hybrid algorithm with a SWAP transition to allow for non-projective parsing (Smith et al., 2018). They also use an static-dynamic oracle where the static oracle is only used for the SWAP transition and is otherwise fully dynamic (de Lhoneux et al., 2017b). They then train their parsers on single treebanks, multiple treebanks for separate languages, and multiple treebanks from related languages. In their multi-treebank models they use a *treebank embedding* which they add to the input of their system. They idea is that although the data is annotated under the same framework there can be differences in annotation and more importantly differences in domain and genre. The models trained on multiple treebanks outperformed parsers trained on single treebanks with the most marked improvement for low-resource and smaller treebanks. There are some other datapoints suggesting that treebank embeddings are beneficial when combining treebanks for monolingual data and for aiding low-resource parsing (de Lhoneux et al., 2017a; Stymne et al., 2018). Multi-lingual work has continued down this path on training or fine-tuning pretrained models on many languages, for example using pre-trained transformer based models fine-tuned on many treebanks (Kondratyuk and Straka, 2019; Üstün et al., 2020).

2.2 Current state of affairs

As final word on the current systems, the current top performing systems on PTB are either pointer networks or graph-based. Ji et al. (2019) used graph neural networks to learn enriched high-order information from partial parses, extending the biaffine system. Zhang et al. (2020a), another leading system on PTB, is the biaffine parser with a second-order CRF for decoding scores. For what it's worth, the current highest scores reported on PTB come from another biaffine parser (Clark et al., 2018). They gain improvements by training a fairly large model on labelled data and a mountain of unlabelled data.

Table 2.1 is not a fair shake considering most are single models, which could be misleading instances of a given model (Reimers and Gurevych, 2017). It is still somewhat indicative for performance on PTB. Although it is a fairly limited comparison as it is for one treebank in English covering one domain. Note that the SeqLab was used with the relative POS tag encoding and so unlike the others it actually needs the POS tags to run. However, while the others don't, they do all use them as input features and so the tagging speed isn't included. Figure 2.3 shows the Pareto front for inference speed and LAS for the parsers we ran locally. Based on this, the pointer networks appear to be the best choice if accuracy is the only priority, however, results in Section 3.4 suggest otherwise.

2.2.1 The rise of PLMs

Interest in dependency parsing as an NLP task in and of itself has diminished in recent times due in large part to the emergence of large pretrained language models (PLMs) being used as contextualised word embeddings. Many studies have observed that these PLMs encode syntactic information and not just surface patterns, but also deeper, hierarchical syntactic structures and that they might even capture the *classical* NLP pipeline (Gulordava et al., 2018; Peters et al., 2018; Tenney et al., 2019; Clark et al., 2019a; Sorodoc et al., 2020). There is also strong evidence that PLMs capture morphosyntactic information, most clearly number agreement between subject and verb (Linzen et al., 2016; Goldberg, 2019;

	speed (sent/s) GPU	CPU	UAS	LAS
Pointer-TD (Ma et al., 2018)	10.2†	-	95.87*	94.19*
Pointer-LR (Fernández-González and Gómez-Rodríguez, 2019)	23.1*	-	96.04*	94.43*
Biaffine w CRF (Zhang et al., 2020a)	400*		96.14*	94.49*
GNN (Ji et al., 2019)	415.9*	-	95.97*	94.31*
HPSG (Zhou and Zhao, 2019)	158.7*	-	96.09*	94.68*
BIST - Transition (Kiperwasser and Goldberg, 2016)	-	76±1‡	93.9*	91.9*
BIST - Graph (Kiperwasser and Goldberg, 2016)	-	80±0‡	93.1*	91.0*
Biaffine (Dozat and Manning, 2017)	411*	-	95.74*	94.08*
CM (Chen and Manning, 2014)	-	654*	91.80*	89.60*
SeqLab (Strzyz et al., 2019b)	648±20*	101±2*	93.67*	91.72*
Distilled-Ensemble (Kuncoro et al., 2016)	-	20*	94.26*	92.06*
UUParser (Smith et al., 2018)	-	42±1	94.63	92.77
Biaffine (PyTorch)	1003±3	53±0	95.74	94.07
SeqLab	1064±13	99±1	93.46	91.49
Pointer-LR	94.8±1.2	8.4±0.0	96.02	94.47
MaltParser 1.9.2 w/ Stack lazy (Nivre et al., 2007)	-	473±11	89.29	86.95

Table 2.1: Speed and accuracy performance for current leading parsers for the English PTB with POS tags predicted from the Stanford POS tagger. ∗ denotes values taken from the original paper, † from Fernández-González and Gómez-Rodríguez (2019), and ‡ from Strzyz et al. (2019b). Values with no superscript (i.e. the second section of the table) are from running the models on our system locally with a single CPU core for both CPU and GPU speeds (averaged over 5 runs) and with a batch size of 256 (excluding UUParser which doesn't support batching) with GloVe 100 dimension embeddings. For Pointer-LR we used the sskip 100 vectors (Ling et al., 2015) as in the original which explains the score being much closer to that reported in the original compared to the other systems we ran natively. **Hardware**: Intel Core i7-7700 and Nvidia GeForce GTX 1080. **Software**: Python 3.7.0, PyTorch 1.0.0, and CUDA 8.0.

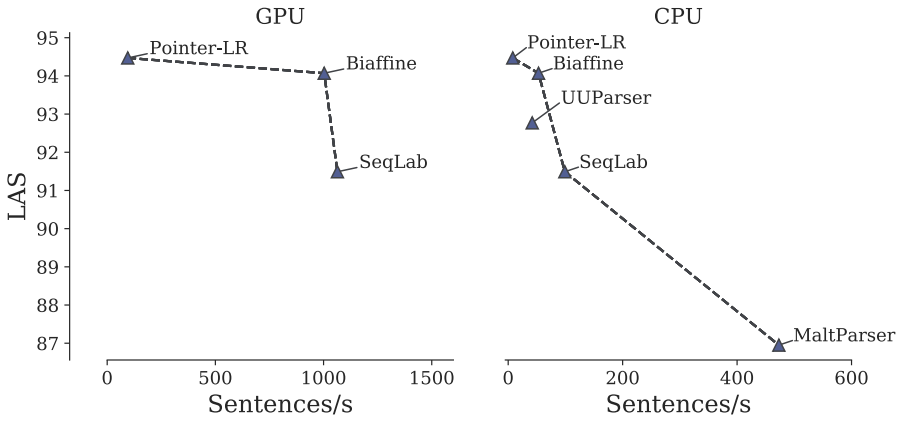


Figure 2.3: Pareto front of modern parsers run on our machine locally.

Lakretz et al., 2019; Giulianelli et al., 2018). Others have observed that they only seem to learn surface level syntactic information and struggle to generalise especially with respect to aspects like licensing of reflexive pronouns and negative polarity and complex subordinate clauses (Futrell et al., 2018, 2019). Although, others have found that they do find information regarding the licensing contexts and corresponding negative polarity items (Jumelet and Hupkes, 2018). The clearest evidence that PLMs haven't solved parsing or syntax is their inability to distinguish between grammatical and ungrammatical constructions (Marvin and Linzen, 2018; Chowdhury and Zamparelli, 2018; Warstadt et al., 2019). More concretely, McCoy et al. (2019) exposed the dependency of PLMs on what they call *"fallible syntactic heuristics."* Further, Vilares et al. (2020) found that without finetuning, PLMs perform poorly at both constituency and dependency parsing. Sinha et al. (2021) obtained results suggesting that PLMs don't depend on word order at all by training models on scrambled text while preserving varying amounts of contextual information (by shuffling sets of words). They still obtain high performance when fine-tuning them for different tasks. Specifically for dependency parsing, when shuffling bigrams the UAS they obtained is only 2 points less on PTB and the UD English-EWT treebank when compared against the PLM trained on the normal data. A specific study on whether PLMs can leverage explicit syntactic information concluded that any positives from using such information are negligible, however, it was relatively limited in scope as to what downstream tasks were evaluated and how the syntactic information was utilised (Glavaš and Vulić, 2021). Others have observed similar results for smaller systems e.g. Fares et al. (2018) summarised the results of parsers at the Extrinsic Parser Evaluation shared task where they found no strong correlation between intrinsic parsing performance and downstream tasks. The systems had little variation (≈ 6 for the three tasks) in performance for event extraction, negation resolution, and opinion analysis for English despite large variation in LAS (> 20). Although it has been observed that an increase in parsing performance doesn't increase downstream tasks after a certain threshold, e.g. when using syntactic information for sentiment analysis (Gómez-Rodríguez et al., 2019). Further, there has been a plethora of work with findings that suggest explicit syntactic signals can help PLMs on downstream tasks even if the impact is not huge (Xu et al., 2020; Li et al., 2020; Zhang et al., 2020c; Bai et al., 2021; Sachan et al., 2021).

What is clear is that these PLMs offer large improvements in many downstream tasks. But even taking this into consideration, not everyone has the resources to deploy these large models. And if a given task can benefit from syntactic information, whether implicitly via PLMs or explicitly from parser systems, it remains much more efficient to use parser systems. Beyond the purely practical reasons, parsing remains of interest as it relates to linguistic focused research inside and outside of NLP. Also, the blackbox nature of current systems is hardly satisfying in a theoretical sense. And then there's always knowledge for knowledge's sake.

PART II

Developing Parsers

Chapter 3

Chunk-and-pass parsing

Work in this chapter is based in part on published work in Anderson et al. (2019).

In this chapter we discuss work influenced by a psycholinguistic model of language comprehension. It theorises that humans understand language via a real-time construction of hierarchical abstractions due to working memory restrictions (Christiansen and Chater, 2016). We look at this specifically for two levels of abstraction with respect to syntax: a shallow parse or chunking which highlights phrase structures and a full dependency parse of a sentence over these phrases. We are not aiming to validate or refute this psycholinguistic model, but are rather using it to guide the development of systems to more efficiently parse sentences. In Section 3.3, we discuss early work we did in order to automatically annotate treebanks with chunk information using an evolutionary algorithm. We also present an evaluation on the quality of these chunks in a number of multi-task experiments. Then in Section 3.4 we present the final system we used in our attempt to use this concept of hierarchical parsing to develop a more efficient parser. We use an information theory process to extract chunks in the final system due to the lengthy training time of evolutionary algorithms. We present results on how our chunk-and-pass system affects the accuracy and speed of a number of modern leading parsing sytems.

3.1 Introduction

We aim to develop an efficient parsing system by taking inspiration from human language comprehension. Humans work under tight cognitive restrictions, not least working memory restrictions (Miller, 1956; Gruszka and Nęcka, 2017). Christiansen and Chater (2016) discuss how the length and speed of auditory linguistic input results in a *now-or-never* bottleneck, namely that the input needs to be processed immediately, otherwise it is lost. They highlight certain repercussions of this limitation and how it relates to how humans process language. The consequence that we are interested in relates to the construction of a real-time hierarchical abstraction of syntactic structures.

This theoretical consideration directed the development of a two-step pipeline for parsing text. The first step creates the first level of *abstraction* by collecting tokens together to form phrases, or chunks words, using a smaller, faster network. These chunks are then used as the input to more complex, robust parsing systems to predict the second level of *abstraction*: the dependencies connecting the chunks. These two levels are then collated to form a full parse tree. The idea is that the lower-level abstracted form compresses the data so that the

slower parsers have less data to process, so the end result is a faster system overall but one that retains more accuracy than merely casting the slower parsers into smaller networks.

Fist, we give a brief discussion on chunking and how it relates to parsing in Section 3.2. In Section 3.3, we introduce a method for automatically annotating treebanks with chunking information using an evolutionary algorithm and how these annotations can potentially aid morphological and syntactic tasks. Having shown that automatically extracting chunks from treebanks is feasible, we then introduce our full chunk-and-pass parsing system in Section 3.4 using a more sophisticated information theory based approach for automatically annotating phrases. We present a series of experiments to evaluate our system when used with three distinct leading parsing systems.

3.2 Related work

Shallow parsing, or chunking, consists of identifying constituent phrases (Abney, 1997). As such, it is fundamentally associated with constituency parsing as it can be used as a first step for finding a full constituency tree (Ciravegna and Lavelli, 1999; Tsuruoka and Tsujii, 2005). Chunking information can also be beneficial for dependency parsing (Attardi and Dell'Orletta, 2008; Tammewar et al., 2015) and vice versa (Kutlu and Cicekli, 2016). Ramshaw and Marcus (1999) introduced the IOB2 tagging scheme for chunking which is used as the basis not only for chunking but also for other spanning tasks such as named entity recognition (Grishman and Sundheim, 1996).

Socher et al. (2010) also look at phrase structures and how they relate to parsing, but their focus is on developing phrase embeddings in conjunction with training parsers. Closer to our work, Kudo and Matsumoto (2002) introduced a statistical dependency parser which was based on chunking data at numerous stages, which improved parsing and training efficiency compared to contemporary systems. Others have used a similar approach but often with accuracy lagging behind the most succesful systems of their day (Sang, 2000; Tsuruoka and Tsujii, 2005). However, Tongchim et al. (2008) obtained a sizeable improvement in their statistical Thai dependency parser when using predicted noun phrase (NP) chunking information.

Tanaka et al. (2017) also consider parsing and chunking together, however, they focus on Japanese in UD where Japanese is tokenised by the short unit word rather than the long unit word, so chunking in this context is almost the reconstruction of words, although some chunks are multi-word expressions. They find the parsers using the long unit words outperform those that use the short unit words and also that the best performance is achieved when combining these two tasks. More recently, Lacroix (2018) explored the efficacy of noun phrase chunking with respect to Universal Dependency (UD) parsing and POS tagging for English treebanks. As UD treebanks do not contain chunking annotation, they extracted chunks by adopting linguistic-based phrase rules. They observed improvements on POS and morphological feature tagging in a shared multi-task framework for the English treebanks in UD version 2.1 (Nivre et al., 2017). However, an increase in performance for parsing was only obtained for one treebank when using the chunk information as direct input.

3.3 Evolutionary algorithm

In this section, we introduce a language-agnostic technique for automatically extracting chunks from dependency treebanks. We evaluate these chunks on a number of morphological and syntactic tasks, namely POS[1] tagging, morphological feature tagging, and dependency parsing. We test the utility of these chunks in a host of different ways. We first consider chunking as one task in a shared multi-task framework together with POS and morphological feature tagging. The predictions from this network are then used as input to augment

[1]POS tagging is used throughout to refer to universal part-of-speech (UPOS) tagging.

sequence-labelling dependency parsing. Finally, we investigate the impact chunks have on dependency parsing in a multi-task framework. Our results from these analyses show that these chunks improve performance at different levels of syntactic abstraction on English UD treebanks and a small, diverse subset of non-English UD treebanks.

3.3.1 Defining chunks

While Lacroix (2018) described a method to obtain chunks from English sentences with UD annotations, their approach is limited to NP chunks and requires hand-crafted linguistic rules, meaning that it cannot be transferred to other languages without language-specific knowledge. In contrast, we introduce a fully automatic approach to obtain chunks from UD-annotated sentences in a language-agnostic way. Figure 3.1 depicts our method of extracting candidate chunk types.

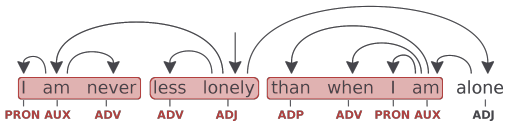


Figure 3.1: Candidate phrase rules are extracted by selecting subtrees with one level of dependency.

Chunk definition Here we loosen the definition of a chunk and consider any base-level subtree a possible chunk defined by the following criteria: (i) the components of a chunk are syntactically linked; (ii) there is only one level of dependency (one head and its dependents); (iii) the components are continuous; and (iv) no dependent within a chunk has a dependent outside the chunk.

Describing chunks with rules For each subtree in the training set that meets the above criteria, the corresponding sequence of POS tags of its words is saved as a candidate rule. Each rule is collected for a given treebank to construct a ruleset of unique candidate chunk types. When more than one overlapping subtree meets these conditions the maximal substring is used, e.g. in Figure 3.1 PRON AUX ADV is chosen instead of PRON AUX or AUX ADV. We allow any chunk type with the exception of those containing the PUNC POS tag and we apply a mild frequency cut of 5 to make the problem more tractable. The English-EWT treebank, for example, results in a ruleset consisting of 512 candidates.

Annotating with rulesets This ruleset (or any subset of it) can be applied to a UD treebank to obtain chunks, by using them as patterns that generate a chunk when they are matched by a sequence of POS tags and meet the criteria described above.[2] In particular, we can apply it to the training set to obtain a set of chunks on which to train a statistical chunker to process arbitrary texts and help morphological and syntactic tasks. When annotating a treebank, the POS tag of the head is used as a suffix for the chunk type, e.g. DET ADJ NOUN would result in IOB tags of B-NOUN and I-NOUN, assuming the head of this phrase corresponds to the NOUN tag (Ramshaw and Marcus, 1999).

However, not all candidate rules are useful and can impact the ability of a chunker to make sensible predictions. For this reason, we will not use the whole candidate ruleset obtained from a training corpus, but instead try to find a subset of the ruleset whose resulting set of chunks strikes a good balance between the following criteria: (i) coverage (i.e. there should be enough chunks to maximize their informativeness for morphological and syntactic tasks) and (ii) consistency and learnability (i.e. the chunks should follow patterns predictable enough to be easily learnable by a machine learning model, so that our approach is not undermined by low chunking accuracy). Our hypothesis is that these two characteristics (which we quantify

[2]Rules are applied from longer (more specific) to shorter (more generic).

with a fitness function in the next section) are reasonable proxies for the usefulness of a particular set of chunks for morphological and syntactic tasks.

Note that to achieve this, it is not possible to merely remove error-prone rules from the ruleset because there is a complicated interplay between rules, i.e. if the 10% most error-prone rules are removed, the overall accuracy of the system is not guaranteed to improve. Furthermore, with so many candidate rules, it is not possible to try every combination as this results in an astronomical number (2^n). Therefore, we aim to use an evolutionary method to find optimal subsets of rules to be used when annotating treebanks.

3.3.2 Evolutionary search for chunk rules

Evolutionary algorithms aim to optimise an objective (fitness) function by evaluating a population of individuals and subsequently generating a new population based on the best performing individuals from the population (Back, 1996). This process is then repeated until a set number of generations is reached or until the fitness function converges. Each individual consists of a set of parameters and its corresponding objective function value, or fitness. The fitness of an individual is used to decide whether to use it as a parent for subsequent generations or to remove it from the population. We introduce the techniques used to select parents and how they are then used to generate offspring (Algorithm 8).

```
for gen ← max_gen do
    for ind in population do
        ind.fit ← GetFitness(ind)
    end for
    offspring ← select(population)
    offspring ← clone(offspring)
    for pair in offspring_2i, offspring_2i+1 do
        if random < P_crossover then
            pair ← Crossover(pair)
        end if
    end for
    for ind in offspring do
        if random < P_mutate then
            ind ← Mutate(ind)
        end if
    end for
    population ← offspring
end for

function GetFitness(ind)
    rules ← Convert(ind)
    train, dev ← ChunkTreebanks(rules)
    TrainChunker(train)
    F1 ← EvalulateChunker(dev)
    Rp ← GetMaxRProportion(dev)
    return F1 + 0.5·Rp
end function
```

Algorithm 8: Evolutionary algorithm.

K-best parent selection The selection operator makes the population converge. We used the simple k-best method where the top k individuals of a population are selected as the parents.

Mutation Mutation is a genetic operator which prevents a population becoming too genetically similar by randomly altering individuals. This ensures that at least some level of genetic diversity is maintained from generation to generation. Our individuals have binary genes, so our mutation operator flips each gene with a probability $P_{\text{mutate gene}}$.

Crossover Crossover is a genetic operator which also preserves genetic variety in a population. In single-point crossover, a random index κ is chosen and the substring 0-κ of parent$_x$ is replaced with the corresponding part of parent$_y$ and vice-versa. This results in two offspring. Single-point crossover can be extended to x-point crossover, where x points are used to cut individuals.

hyperparameter	value
population size	100
number of generations	4
k-best	5
P_{mutate}	0.5
$P_{\text{mutate gene}}$	0.05
$P_{\text{crossover}}$	0.5
decay (linear)	0.1

Table 3.1: Hyperparameters for the evolutionary algorithm: k-best, the number of best parents chosen to seed next generation; P_{mutate}, the probability an individual will mutate; $P_{\text{mutate gene}}$, the probability a given gene will mutate; $P_{\text{crossover}}$, the probability a pair of individuals will crossover; and decay is how much P_{mutate} and $P_{\text{crossover}}$ decrease after each generation.

We used the DEAP framework for our implementation (Fortin et al., 2012) with the parameters in Table 3.1. We represented our rulesets as a binary vector, where 1 meant a rule was used and 0 meant it was not. Our fitness function was obtained by combining the F1-score of a chunker implemented with the sequence-labelling framework NCRF++ (Yang and Zhang, 2018) and the proportion of the maximum compression rate, weighted 1.0 and 0.5 respectively. The compression rate, r, is defined as:

$$r = \frac{C_{\text{tokens}}}{C_{\text{chunks}} + C_{\text{out}}} \tag{3.1}$$

where C_{tokens} is the number of tokens in a treebank, C_{chunks} the number of chunks a ruleset creates, and C_{out} the number of tokens outside of chunks. And subsequently the proportion of the maximum compression rate, $r\%$ is defined as:

$$r\% = \frac{r_{\text{subset}} - 1}{r_{\text{all}} - 1} \tag{3.2}$$

where r_{subset} is the compression rate of the current rule subset and r_{all} is the compression rate of the full ruleset.

We used a small network for chunking due to the considerable computational costs of evolutionary algorithms. For each individual in each population, we trained a chunker for 5 epochs (see Table 3.2 for the parameters) and the corresponding model's best performance on the development set was taken as that individual's fitness along with the proportion of the maximum compression rate, $r\%$: the proportion of the maximum rate was used to prevent the algorithm from generating rulesets that generated few chunks and therefore minimising the potential impact. The convergence over 40 generations for English-EWT and Japanese-GSD can be seen in Figure 3.2.

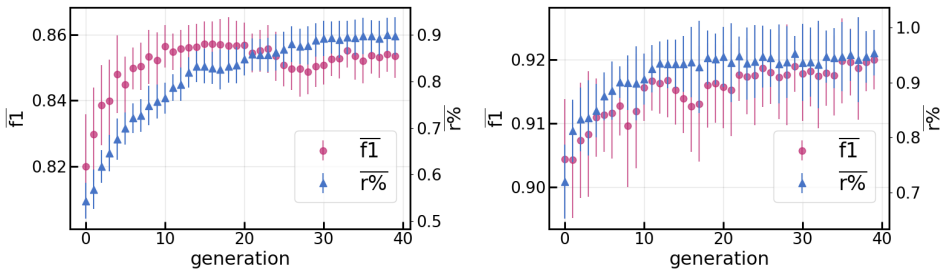
As a final step, we took the top 100 best rulesets from across generations and extracted the rules that appeared in at least 75% and 95% of these sets, as the evolutionary algorithm only managed to find a single set with a fairly low performance. Rulesets were obtained this way for each treebank, except the rulesets extracted from English-EWT were subsequently used on the other English UD treebanks. The statistics for the resulting chunks for the respective test data can be seen in Table 3.3.

3.3.3 Sequence-labelling MTL framework

All the proposed tasks can be cast as sequence labelling, so in this work we have used a sequence-labelling framework to address them. The framework is built on BiLSTM layers.

hyperparameter	value
BiLSTM dimensions	200
BiLSTM layers	1
word embedding dimensions	50
character embedding dimensions	30
character hidden dimensions	50
character CNN layers	4
CNN window size	3
optimiser	SGD
loss function	cross entropy
learning rate	0.015
decay (linear)	0.05
momentum	0.9
dropout	0.5
L_2 regularisation	$1x10^{-8}$
epochs	5
training batch size	10
runtime batch size	128

Table 3.2: Hyperparameters for the neural chunker used during the evolutionary algorithm.



(a) English-EWT (b) Japanese-GSD

Figure 3.2: Average F1-score and proportion of max compression for English-EWT (a) and Japanese-GSD (b) during evolutionary search for optimal chunk type candidates.

	# rules		C/sent	
	75%	95%	75%	95%
English-EWT	230	134	3.11	2.71
English-GUM	-	-	4.48	3.84
English-Lines	-	-	4.47	4.18
English-ParTut	-	-	6.32	5.84
Bulgarian-BTB	152	108	3.94	3.65
German-GSD	135	106	4.05	3.90
Japanese-GSD	184	130	6.83	6.70

Table 3.3: Chunking statistics on test data for each treebank used where # rules is the number of rules in a ruleset for a given threshold and C/sent corresponds to the number of chunks per sentence found.

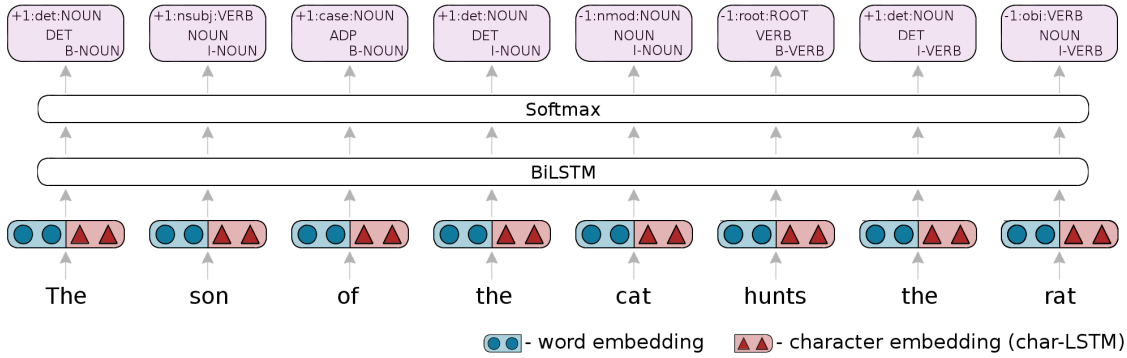


Figure 3.3: Multi-task architecture shown with sequence-labelling dependency parsing (as described in subsection 3.3.3), POS tagging, and chunking as shared tasks. Network input is a concatenation of word embeddings (circles) and character-level word embeddings (triangles) obtained from a character-based LSTM layer. The network is constructed of BiLSTM layers followed by a *softmax* layer for inference.

The input to the network are continuous word representations and character embeddings. For this work, we used NCRF++ (Yang and Zhang, 2018), which uses stacked BiLSTMs, to generate contextualised hidden representations for every word ($\vec{h_i}$) in the input sentence. For decoding, it uses a feed-forward layer followed by a *softmax* activation:

$$P(y|\vec{h_i}) = softmax(\vec{W} \times \vec{h_i} + \vec{b}) \tag{3.3}$$

The single task models are optimised with cross-entropy loss, $\mathcal{L}$, defined as:

$$\mathcal{L} = -\sum log(P(y|h_i) \tag{3.4}$$

For the multi-task learning models, we implemented a hard-sharing architecture, where all the stacked BiLSTMs are shared across all tasks (Søgaard and Goldberg, 2016). A separate feed-forward layer (as the one used in the single task setup) is used to decode the output for each task. With respect to the computation of the loss under the multi-task learning (MTL) setup, $\mathcal{L}_{MTL}$, is defined as:

$$\mathcal{L}_{MTL} = \sum_{t \in T} \beta_t \mathcal{L}_t \tag{3.5}$$

where t is a task from the set of all tasks, T; β_t is the corresponding weight for task t; and $\mathcal{L}_t$ is the cross-entropy loss for task t. A schematic of the network can be seen in Figure 3.3.

Dependency parsing as sequence labelling In order to more readily utilise the multi-task framework for dependency parsing, we have cast dependency parsing as a sequence-labelling task. This was done by using the relative position encoding scheme introduced by Spoustová and Spousta (2010). We opted to use this encoding as it was the highest performing labelling scheme evaluated in Strzyz et al. (2019b). For each word in a sentence the dependency relation label is combined with the relative position of its head based on the POS tag of the head, e.g. a noun which is the subject of a verb (*son* in the input sentence in Figure 3.3) would have a label of +1,nsubj,VERB, where +1 indicates the head is the next VERB in the sentence and nsubj is the relation label.

3.3.4 Experiments

Data The analyses were undertaken using the English treebanks (EWT, GUM, LinES, and ParTUT) and also Bulgarian-BTB, German-GSD, and Japanese-GSD from UD v2.3 (Nivre et al., 2018). No results are given for Japanese-GSD for morphological feature tagging as it does not contain this information.

Network hyperparameters We used the framework as described above and hyperparameters from Vilares et al. (2019) which can be seen in Table 3.4. The standard input to the system consisted of word embeddings concatenated with character embeddings. All embeddings were randomly initialised.

hyperparameter	value
BiLSTM dimensions	800
BiLSTM layers	2
word embedding dimensions	100
character embedding dimensions	30
character hidden dimensions	50
feature dimensions	20
optimiser	SGD
loss function	cross entropy
learning rate	0.2
decay (linear)	0.05
momentum	0.9
dropout	0.5
epochs	100
training batch size	8
runtime batch size	128

Table 3.4: Hyperparameters for the network used in all experiments.

Experiment 1 We tested the impact of our chunks on POS and morphological feature tagging in a shared multi-task setting. This entails feeding word and character embeddings as input to the network with the output being some combination of POS tags, morphological feature tags, and chunk labels. These results were compared against the baseline taggers (single-task networks and POS and morphological features shared only). Tasks were equally weighted. As a further baseline we include results for POS and morphological feature tagging using UDPipe 1.2 (Straka and Straková, 2019a).

Experiment 2 We used the best predictions (when using chunking) from experiment 1 as additional features for a sequence-labelling dependency parser (Strzyz et al., 2019b). Therefore, network input consisted of word and character embedding and then some combination of POS tags, morphological feature tags, or chunk labels with the sole output being a dependency parser tag. We used gold tags and labels as input during training, but at runtime we used predicted tags and labels. For baselines we train a model with no features which is decoded with predicted POS tags using UDPipe 1.2 (as the sequence-labelling encoding we are using requires POS tags to resolve dependency heads) and also a model trained with POS tags as features but also using UDPipe 1.2 predicted POS tags at runtime.

Experiment 3 We tested the impact of our chunks on a sequence-labelling dependency parser in a multi-task framework with and without the other tasks. POS tagging was treated as a secondary main task with a weight of 0.5 (as POS tags are needed to decode the sequence-labelling scheme for the dependency parser) and chunks and morphological features were considered auxiliary tasks with a weight of 0.25 when used. The input during this experiment were only word and character embeddings. An example is shown in Figure 3.3 where the shared tasks are chunking, POS tagging, and dependency parsing. The baseline used here is a model trained solely to predict dependency parsing tags which are then decoded using predicted POS tags from UDPipe 1.2.

3.3.5 Results and discussion

As seen in Tables 3.5 and 3.6 the multi-task framework with chunks improves the performance of both POS and morphological tagging for all English treebanks. In the same table, it is clear that they do not aid Bulgarian, but they do improve POS tagging performance for German

	EWT		GUM		Lines		ParTut	
	UPOS	Feats	UPOS	Feats	UPOS	Feats	UPOS	Feats
UDPipe	94.44	95.37	93.88	94.21	94.73	94.83	94.10	94.01
Single	95.08	96.09	94.61	94.92	95.64	95.57	94.69	94.54
pos+feats	95.23	96.21	94.60	95.26	95.59	95.71	94.63	94.16
pos+feats+chunks$_{75}$	**95.89**	**96.72**	**95.58**	**96.31**	**96.38**	**96.45**	96.04	95.51
pos+feats+chunks$_{95}$	95.86	96.52	95.52	96.21	96.35	96.33	**96.21**	**95.60**

Table 3.5: Multi-task tagging performance on English UD treebanks (en-ewt, en-gum, en-lines, and en-partut): single, single-task training; pos, with POS tagging; feats, with morphological feature tagging; and chunks$_x$, with chunks with threshold x.

	Bulgarian-BTB		German-GSD		Japanese-GSD	
	UPOS	Feats	UPOS	Feats	UPOS	Feats
UDPipe	**97.78**	**95.55**	92.03	70.18	96.39	-
Single	97.41	95.06	93.07	87.14	96.97	-
pos+feats	97.69	94.84	92.90	**87.28**	-	-
pos+feats+chunks$_{75}$	97.49	94.58	**93.34**	87.03	96.98	-
pos+feats+chunks$_{95}$	97.44	94.45	92.90	87.11	**97.09**	-

Table 3.6: Multi-task tagging performance on Bulgarian-BTB, German-GSD, and Japanese-GSD UD treebanks: single, single-task training; UPOS, with POS tagging; Feats, with morphological feature tagging (except Japanese-GSD which has no morphological features); and chunks$_x$, with chunks with threshold x.

	Baseline		Multi	
	75%	95%	75%	95%
English-EWT	89.99	91.59	91.84	92.98
English-GUM	85.76	88.11	88.08	89.98
English-Lines	86.01	88.38	88.45	90.67
English-ParTut	88.36	90.78	91.79	93.30
Bulgarian-BTB	92.27	92.60	93.79	94.45
German-GSD	88.74	88.97	89.35	89.62
Japanese-GSD	93.35	92.73	94.39	94.02

Table 3.7: Chunker F1 scores in multi task setting where the baseline presented is from training the chunker for a given ruleset with threshold 75% or 95% as a single task and multi is from training with pos and morphological feature tagging except for Japanese-GSD.

	EWT		GUM		Lines		ParTut	
	UAS	LAS	UAS	LAS	UAS	LAS	UAS	LAS
no features^{udpipe}	80.97	77.87	76.70	72.71	76.43	71.87	81.63	78.67
pos^{udpipe}	84.88	81.79	81.09	76.87	79.06	74.08	84.01	80.63
pos	86.15	83.29	**83.03**	**79.31**	80.76	76.12	85.83	82.69
pos-feats	86.32	83.37	82.83	79.13	**81.15**	**76.48**	86.71	83.60
pos-chunks$_{75}$	85.84	82.87	82.49	78.83	80.86	76.04	87.03	83.86
pos-chunks$_{95}$	85.80	82.86	81.95	78.19	80.32	75.55	86.65	83.36
pos-feats-chunks$_{75}$	**86.43**	**83.41**	82.61	78.86	81.13	76.21	87.09	83.86
pos-feats-chunks$_{95}$	85.99	83.04	82.15	78.50	80.82	76.09	**87.35**	**84.04**

Table 3.8: Feature input ablation for dependency parser with English UD treebanks (en-ewt, en-gum, en-lines, and en-partut): no features^{udpipe}, no features but UDPipe predicted POS tags used to decode; pos, gold POS tags for training and predicted POS tags for runtime (pos^{udpipe} UDPipe predicted POS tags used); feats, gold morphological feature tags for training and predicted feature tags for runtime; and chunks$_x$, gold chunks with threshold x at training time and predicted chunks for runtime.

	Bulgarian-BTB		German-GSD		Japanese-GSD	
	UAS	LAS	UAS	LAS	UAS	LAS
no features^{udpipe}	86.49	82.43	63.20	58.86	89.96	88.43
pos^{udpipe}	89.48	85.30	79.39	74.04	92.49	90.42
pos	89.47	85.11	81.77	76.69	**93.68**	**91.70**
pos-feats	**89.74**	**85.48**	**82.05**	**77.12**	-	-
pos-chunks$_{75}$	89.23	84.67	81.49	76.54	93.28	91.41
pos-chunks$_{95}$	89.06	84.77	81.55	76.40	92.95	91.20
pos-feats-chunks$_{75}$	89.11	84.83	81.77	76.71	-	-
pos-feats-chunks$_{95}$	89.24	85.07	81.41	76.38	-	-

Table 3.9: Feature input ablation for dependency parser with Bulgarian-BTB, German-GSD, and Japanese-GSD UD treebanks: no features^{udpipe}, no features but UDPipe predicted POS tags used to decode; pos, gold POS tags for training and predicted POS tags for runtime (pos^{udpipe} UDPipe predicted POS tags used); feats, gold morphological feature tags for training and predicted feature tags for runtime; and chunks$_x$, gold chunks with threshold x at training time and predicted chunks for runtime.

	EWT		GUM		Lines		ParTut	
	UAS	LAS	UAS	LAS	UAS	LAS	UAS	LAS
single^{udpipe}	80.97	77.87	76.70	72.71	76.43	71.87	81.63	78.67
pos	84.52	81.30	78.94	74.96	78.75	74.13	83.66	80.25
pos-feats	84.21	81.14	79.51	75.42	78.56	73.87	84.10	81.31
pos-chunks$_{75}$	**84.55**	**81.51**	79.54	75.48	78.17	73.55	83.86	81.13
pos-chunks$_{95}$	84.42	81.34	79.60	75.54	78.72	**74.20**	83.57	80.16
pos-feats-chunks$_{75}$	84.25	81.24	**79.81**	**75.84**	78.75	73.95	84.01	80.90
pos-feats-chunks$_{95}$	84.24	81.18	79.48	75.36	**78.84**	74.15	**84.98**	**81.92**

Table 3.10: Multi-task parsing results for English (en-ewt, en-gum, en-lines, and en-partut): single^{udpipe}, parsing as single task with UDPipe predicted POS tags used to decode parser output; pos, with POS tagging as aux. task; feats, with morphological feature tagging as aux. task; and chunks$_x$, with chunking as aux. task for threshold x.

	Bulgarian-BTB		German-GSD		Japanese-GSD	
	UAS	LAS	UAS	LAS	UAS	LAS
single^{udpipe}	86.49	82.43	63.20	58.86	89.96	88.43
pos	88.00	83.89	80.75	75.59	**93.25**	91.45
pos-feats	88.07	83.89	80.46	75.50	-	-
pos-chunks$_{75}$	87.90	83.66	**81.29**	**75.96**	**93.25**	**91.61**
pos-chunks-$_{95}$	88.07	83.93	80.98	75.71	93.04	91.28
pos-feats-chunks$_{75}$	**88.26**	**84.00**	80.77	75.52	-	-
pos-feats-chunks$_{95}$	88.09	83.67	80.69	75.63	-	-

Table 3.11: Multi-task parsing results for Bulgarian-BTB (bg), German-GSD (de), and Japanese-GSD (ja) UD treebanks: single^{udpipe}, parsing as single task with UDPipe predicted POS tags used to decode parser output; pos, with POS tagging as aux. task; feats, with morphological feature tagging as aux. task; and chunks$_x$, with chunking as aux. task for threshold x.

and Japanese. Table 3.7 shows that chunking performance consistently improves in the multi-task setting. Parsing performance is improved across all treebanks when the predictions

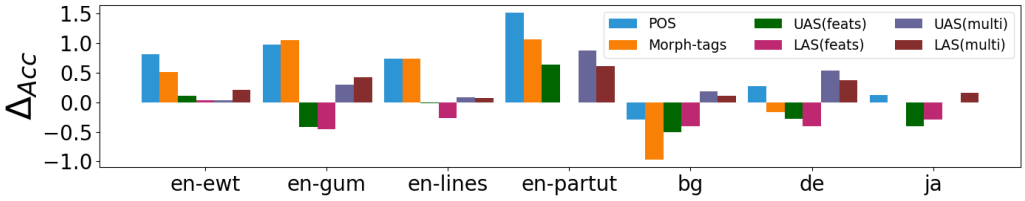


Figure 3.4: Impact of chunks on other tasks.
Difference in accuracy for each task between the best model with chunks and the best without.

from experiment 1 are used as features (Tables 3.8 and 3.9), but only for English-EWT (the largest treebank) and ParTUT (the smallest) do the predicted chunks explicitly improve performance and for the other treebanks only the other predicted features help. This is in contrast to the findings of Nguyen and Verspoor (2018), who obtained higher performance for larger treebanks. In the multi-task setting for the dependency parser (Tables 3.10 and 3.11), the chunking information consistently aids performance with a meaningful increase in accuracy observed over baseline models for each treebank.

As can be seen in Figure 3.4, the change in performance when using the predicted chunks as a feature for parsing is less profound than the results from the multi-task experiments. Only two English treebanks explicitly benefit from predicted chunks, whereas all treebanks benefit from at least one feature. So the performance is at least implicitly improved by using our chunks, except for the more morphologically-rich (especially with respect to verbal inflection) Bulgarian. The treebank used for Japanese, generally an agglutinative language, does not contain morphological features, so perhaps it too would not improve with chunks if they could have been used. Therefore, it would be interesting to evaluate whether the impact of chunking information is predicated by certain linguistic features. Furthermore, the increase in performance for each treebank for the multi-task experiments suggests that the performance when using the chunks as input would improve with better predicted chunks, which corroborates the findings of Lacroix (2018).

3.3.6 Summary

We have introduced a language-agnostic method for extracting chunks from dependency treebanks. We have also shown the efficacy of these chunks with respect to improving POS tagging, morphological feature tagging, and dependency parsing for a number of English and non-English UD treebanks. Furthermore, as our technique can be applied to any treebank, the impact of chunking information on morphological and syntactic tasks across a broad range of languages can readily be investigated.

3.4 Chunk-and-Pass

Having established that it is feasible to automatically extract chunks that are useful for syntactic and morphological analysis, we establish a less computationally expensive means of extracting chunks by using information theory. We use normalised pointwise mutual information (NPMI) over POS tags and dependency relations to effectively score candidate rules for extracting chunks. We then use these rules to *chunk* treebanks and use them to train a Chunk-and-Pass pipeline. First a chunker is trained to predict what should be chunked and what should not be chunked while also predicting how to combine the elements of chunks with respect to dependency relations. Then a standard parsing system is used to train the outside element which learns how to combine the chunks into a dependency tree. Then the predictions of these systems are collated to form a full parse. We do this for

a number of different settings: different size of network for the chunkers, different levels of compression, different treebanks, and different OUTSIDE parsers.

3.4.1 Methodology

Here we describe the methodology used for creating chunked data for training the chunker and outside parsers, for creating chunked embeddings for the outside parsers, and our out-of-vocabularly (OOV) treatment of chunks. We also describe the outside parsers and treebanks we used.

NPMI for selecting chunks For each node n in a tree t with a head at position $n+k$, where k is an integer and can be negative or positive, we evaluate whether this is a valid phrase candidate by checking whether all nodes between n and $n+k$ share the same head as n. We do not consider subphrases of potential phrases, such that once a sequence of nodes are considered part of a phrase, the next node to be checked is the head of that phrase.

This process results in conflicting definitions of a phrase, e.g. nodes 1 to 3 could be considered a phrase with the head of the phrase at node 3 and then nodes 3 to 5 could be considered a phrase with node 5 as the head of the phrase. We order the potential phrases in a given tree by the average NMPI over the nodes in a phrase and we select the phrases which have the highest average NMPI first and remove any potential conflicting phrases from the candidacy list.

We use NMPI so as to set more intuitive thresholds as the value lies between -1 and 1 (Bouma, 2009). We use the NMPI of UPOS tag of a node, t, and the tuple of the UPOS of head of that node, h, and the relation between the node and its head, rel. Such that for a given node:

$$\text{MPI}(t;h,rel) = \log \frac{p(t;h,rel)}{p(t)p(h,rel)} \tag{3.6}$$

$$\text{NMPI}(t;h,rel) = \frac{\text{MPI}}{-log(p(t;h,rel)} \tag{3.7}$$

The average NMPI is then:

$$\langle \text{NMPI} \rangle = \frac{1}{N-1} \sum_{d \in C} \text{NMPI}(t_d;h_d,rel_d) \tag{3.8}$$

where N is the number of nodes in a phrase and d is a dependent in a phrase C.

We then can set the threshold on the average NMPI to select which phrases are labelled as chunks. We use an extension of the IOB2 tagging scheme, where the begining of a chunk is labelled as B, the inside is labelled as I, and tokens which are not part of a chunk are labelled as O. We append this with the *chunk type*, which is just the UPOS of the head of a chunk, and with the relation of the node to its head. So for a phrase such as *the bluttered loon* where *loon* is the head and has UPOS tag of `NOUN`, *the* has UPOS tag `DET` and is connected to the head via an edge labelled `det`, and *bluttered* is an `ADJ` with an `amod` relation, the resulting chunk labels would be `B-NOUN|det`, `I-NOUN|amod`, and `I-NOUN|HEAD`. The relation of the head of a chunk is replaced by `HEAD`. The tag for tokens outside of chunks is `O|out`. The relation is appended so that the system developed to predict chunks can also predict the relations of dependents in a chunk, effectively parsing the lowest level structures in a dependency tree. We use training and development data to calculate NMPI values and set an upper limit of 10 tokens in a chunk (this is mainly because the OOV handling procedure as described below becomes very computationally expensive for long chunks).

Compression ratio We also evaluate the impact chunking treebanks has with respect to compression as this gives us an objective metric that is independent of hardware constraints

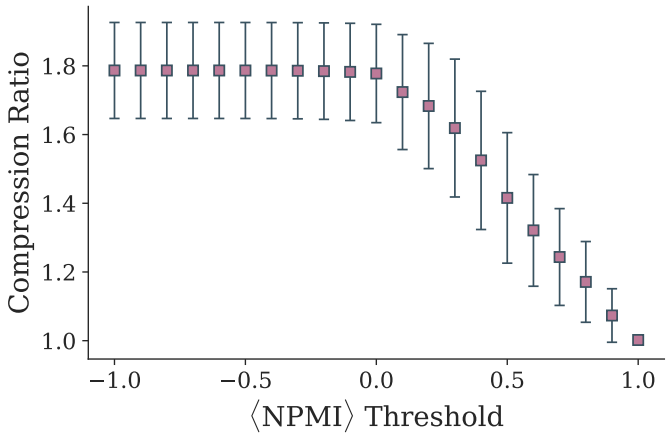


Figure 3.5: Average compression ratio against ⟨NPMI⟩ across all treebanks in UD v2.7 for training and development data. Error bars are the standard deviation.

and implementation choices, which impact measurements of inference speed and energy consumption. It is defined as:

$$\text{compression ratio} = \frac{N_{\text{tokens}}}{N_{\text{out}} + N_{\text{chunks}}} \tag{3.9}$$

where N_{tokens} is the total number of tokens in a sample, N_{out} is the number of tokens not in a chunk, and N_{chunks} the number of chunks.

Chunk embeddings We concatenate the items of a chunk by using the word and character embeddings of each form and averaging them. The form for a phrase such as *the dirty dog* becomes `the|dirty|dog`. We also include all possible combinations where tokens could be missing, e.g. `the|dirty|UNK`, `the|UNK|dog`, `UNK|dirty|dog` and also `UNK|UNK|dog` and so on. This is done to offset OOV issues. Unitary chunks are saved as single words and each word in a chunk is saved as a unitary chunk too.

Handling OOV chunks A major issue with using chunk embeddings as input to a system is out-of-vocabulary chunks. Table 3.12 shows the percentage of OOV chunks for the full form of chunks which are incredibly high, ranging from 10.3% to 50.5%. Also shown are the OOV percentages after using the following process, where the values range from 0.0% to 6.0%.

First, we get all the possible combinations of the forms for the nodes in a chunk, using only the forms that occurred in the training data (because no combination exists with a word not seen already). These combinations include `UNK` tokens and the forms in a chunk, e.g. if `nó|ba|úrlabhraí` and *úrlabhraí* were OOV, the combinations would be `UNK|UNK|UNK`, `UNK|ba|UNK`, `nó|ba|UNK`, and `nó,|UNK|UNK`. To choose one from this list, we calculate the dependency distance of the seen tokens and allocate a distance for the unknown tokens equal to the length of the chunk (basically, this has to be a value higher than the highest possible distance in the chunk). We then use these to calculate the mean dependency distance of each chunk, e.g. 2.67: `nó|UNK|UNK`, 2.3: `UNK|ba|UNK`, 2.0: `nó|ba|UNK`, and 3.0: `UNK|UNK|UNK`. Then the combinations are ordered based on their mean dependency distance and the one that first appears in the vocab is picked. So for the combinations from the example, the chunk form chosen would be `nó|ba|UNK` but if this chunk didn't occur in the vocabulary, then `UNK|ba|UNK` would be chosen. When two combinations have the same mean distance, we choose the combination with the least number of unknown tokens. If that number is equal, we select the one with the head token lexicalised. Failing that, we pick one at random.

We save this chunk embedding form to utilise those learnt at training time from the chunker and we also save the full form of the nodes in a chunk, such that the OUTSIDE

parser would see UNK|ba|UNK for looking up the chunk embedding space but would also see nó|ba|úrlabhraí to make use of the character embedding space of the outside parser.

Sometimes a chunk is predicted that is longer than what is observed in the training data. To handle this we remove the last token in a chunk and follow the same procedure as above.

	r=1.2		r=1.4		r=1.6	
	Full	Partial	Full	Partial	Full	Partial
zh	10.3	0.0	17.6	0.1	23.8	0.2
hi	12.6	0.2	16.7	0.3	23.4	0.6
ko	36.3	0.8	44.4	3.0	50.5	6.0
pl	20.9	0.2	28.6	0.4	35.8	0.7

Table 3.12: OOV percentage on test data where Full means the complete chunk (i.e. no UNK component is included) and Partial means some combination of the forms of a chunk are found (i.e. form is not completely composed of UNK components) for different compression levels (r=x).

Post-processing predicted chunk labels Predicted chunks are not always sensible. So we apply some heuristics to fix problems or redundancies. As defined, all nodes in a chunk are headed by the root of the chunk (except the root), so when a chunk contains more than one node predicted as the root of that chunk, the last node predicted is selected (subsequent predicted roots basically overwrite previous ones). Nodes predicted as the root of a chunk but not selected then have no predicted relation. If w_i is predicted as the beginning of a chunk but so to is w_{i+1}, w_i is treated as a unitary chunk. Similarly, if w_i is predicted as I-rel but w_{i-1} is not B-rel, w_i is treated as a unitary chunk.

Pipeline First, a treebank is labelled with chunk tags based on the rules used for a given NPMI threshold. This labelled data is then used to train a chunker which predicts chunk tags and the relations for the dependents in each chunk. The labelled data is then converted to a compressed version where the chunks replace words and is used to train an outside parser. Then at inference the chunker is used to predict the chunk labels and the relations of chunks. The predicted chunk labels are used to create a compressed version of data which is used by the outside parser. The predictions of the chunker and outside parser are then collated to form a full parse tree.

3.4.2 Chunker system

We use a MTL BiLSTM network for our chunker. We use a set of parameters for the BiLSTM layers so that the network is fairly small. The chunk tags and relations were learnt together as separate tasks. Each was predicted by a separate MLP that was fed the hidden representations from the BiLSTM layer. The loss from both tasks were combined equally and model selection was based on the average accuracy across both tasks. We use a combination of layers and nodes to evaluate the impact the size of network as on the chunker performance: we use either 2 or 3 BiLSTM layers and with 200, 400, or 600 nodes, resulting in 6 network sizes. We also use three compression rates (1.2, 1.4, and 1.6). So for each treebank we train 18 chunkers. Input to the networks is randomly initialised character and word embeddings. The other hyperparameters are the same as those in Table 3.4. For the main results, we don't use the OOV handling procedure as this slows the chunkers down to the point that they are slower than the outside parsers. We evaluate its impact in a secondary experiment.

3.4.3 Outside parsers

All the outside parsers use BiLSTMs in their structure and have structures in addition which set them apart from one another and use one of three paradigms: broadly speaking, one is a transition-based parser, one is a sequence-labelling parser, and the other is a graph-based parser. These are covered in more detail in Chapter 2, but we give a brief recap here for

	Training				Development				Test			
	Sents.	Tokens	Avg. Len.	NP	Sents.	Tokens	Avg. Len.	NP	Sents.	Tokens	Avg. Len.	NP
Hindi	13K	281K	22.1	2.6	2K	35K	22.2	2.4	2K	35K	22.2	2.4
Korean	23K	296K	13.9	4.5	2K	25K	13.2	4.7	2K	28K	13.4	4.0
Polish	18K	282K	16.9	1.4	2K	35K	16.7	1.5	2K	34K	16.2	1.4
Chinese	15K	408K	28.2	0.0	2K	51K	28.0	0.0	2K	49K	27.3	0.0

Table 3.13: Statistic for treebanks used in Chunk-and-Pass experiments.

ease of reference. Each parser is trained with randomly initialised character embeddings and pretrained FastText embeddings (Grave et al., 2018). Except for Chinese, as the FastText embeddings are in the traditional script, so we use the embeddings from Li et al. (2018a).[3]

Left-to-right pointer network (L2R). One of the current top-performing parsers on PTB, it uses a left-to-right transition-based algorithm that builds a number of attachments equal to sentence length using a pointer network (Ma et al., 2018; Fernández-González and Gómez-Rodríguez, 2019).[4]

Deep biaffine (Biaffine) (Dozat and Manning, 2017) is an edge-factored graph-based parser that produces a matrix of scores giving a probability distribution over heads, where the CLE algorithm (Algorithms 2 and 3 in Chapter 1) is then applied to obtain a tree.[5]

Sequence labelling parser (SeqLab) encodes trees as a sequence of labels, so that a direct one-to-one prediction can be made for each token in a sentence (Spoustová and Spousta, 2010; Li et al., 2018b; Strzyz et al., 2019b).[6] We implement it using the Biaffine system described above (for uniformity) editing it to be a sequence-labelling system.

3.4.4 Data

We use a small sample of treebanks covering languages from 3 different language families and 4 sub-families and which represent different syntactic systems covering analytic, fusional, and agglutinative languages. All are written in different scripts. We offer a brief description of the treebanks used and some of the salient features of their respective languages. The treebanks were chosen to represent varying syntactic features, but also because of their high quality from being either manually annotated or manually corrected. We also chose relatively large treebanks to better evaluate the impact of compressing data with our chunk and pass procedure. The statistics for each treebank are shown in Table 3.13.

UD Hindi-HDTB (Hindi) is a UD treebank for Hindi based on manually annotated news data (Palmer et al., 2009; Bhat et al., 2017). Hindi is a lightly fusional language with some degree of verbal inflection and noun declension but also makes extensive use of postpositions (McGregor, 1977). It is a split-ergative language, meaning in certain cases it uses a nominative-accusative structure but in others uses an ablative-ergative syntax where the subject of an intransitive verb behaves like the object of a transitive verb (Comrie, 1978). It also exhibits tripartite behaviour in certain clauses, where the subject of intransitive verbs, the object of transitive verbs, and the subject of transitive verbs all have different case markings (Comrie, 1978). It is nominally a SOV language, but it has a fairly free word order (Snell and Weightman, 1989). It is an Indo-Iranian language written in the Devanagari script.

UD Polish-PDB (Polish) is the largest Polish UD treebank manually annotated on fiction, non-fiction, and news data (Wróblewska, 2018). Polish is a highly fusional language with a

[3] https://jima.me/open/cwv/
[4] https://github.com/danifg/SyntacticPointer.
[5] https://github.com/zysite/biaffine-parser.
[6] We use refactored encoding/decoding functions from https://github.com/mstrise/dep2label.

high degree of verbal inflection (Feldstein, 2001) and 7 case-markings (Wiese, 2011). It is a null-subject language (Cognola and Casalicchio, 2018) with a nominal SVO order but has relatively free word order (Siewierska, 1993). Like most Slavic languages it doesn't make use of articles (Bielec, 1998) but it does have a complex system of numerals and quantifiers that result in agreement mistmatches (Klockmann, 2012). It is a Balto-Slavic language written in the Latin script.

UD Korean Kaist (Korean) is a large treebank generated from a constituency treebank which was semi-automatically annotated with manual corrections based on academic, fiction, and news data (Choi et al., 1994; Chun et al., 2018). Korean is a strongly suffixing agglutinative language (Ramstedt, 1968; Sohn, 1999). This results in a large number of cases and a high degree of verbal inflection (Chang, 1996; Song, 1988; Lee and Ramsey, 2000). It is technically a SOV ordered language but it has a highly flexible word order (Ramstedt, 1968; Sohn, 1999). Korean also uses honorifics and speech levels, the former encoding the social relationship between the speaker and the referents in a discussion and the latter the speaker and the person/people being spoken to (Brown, 2015). It is a Koreanic language written in the Hangul script.

Chinese Penn Treebank (Chinese) is large manually annotated treebank for Mandarin based on news data (Xue et al., 2002, 2005). Chinese is an analytic, isolating language with an SVO dominant word order and is a pro-drop language (Li and Thompson, 1981). Chinese has no grammatical tense markers but relies on context or temporal expressions, but aspect is expressed via the use of particles (Liu, 2015). Classifiers and measure words must be used when a noun is preceded by a number, a demonstrative pronoun, or certain quantifiers which are particles that appear between these qualifiers and their respective nouns (Her and Hsieh, 2010). Chinese is said to be a verb stacking language, where more than one verb or verb phrases are stacked together in the same clause, but there is some disagreement if the way verbs are combined actually constitutes verb stacking (Li and Thompson, 1981; Paul, 2008). It is a Sino-Tibetan language and the data in this treebank is written in simplified Hanzi.

3.4.5 Experiments

All speeds reported here are based on the time models take to parse data at prediction time. We don't include model loading time or the time to save to file, as these overheads can be mitigated by keeping models in memory and avoiding unnecessary writing to disk by running everything together. The times for L2R do include writing to file because the original implementation entangled writing to file and running inference. We measured the impact this has and it is negligible (<1s) and the parser is really slow anyway, so the relative impact is tiny.

Baselines First we establish baselines. In order to evaluate whether Chunk-and-Pass parsers are efficient, we need to know the performance of the outside parsers trained normally but with varying network sizes. In other words, our method has to be both more accurate and faster than if we just trained smaller versions of the standard parsers. Figure 3.6 shows the Pareto front for the outside parsers trained on the full treebank as normal for different numbers of BiLSTM nodes and layers. We tested networks with either 2 or 3 BiLSTM layers and with 400, 600, 800, or 1000 nodes. The overall Pareto front, which is made up of the Biaffine front and part of the SeqLab front, is what we want to improve. We use the largest network setting of 3 layers and 1000 nodes as the version for the Chunk-and-Pass system, as this is on the front and we assume is most likely to perform best on the compressed data.

Note we could effectively choose any network size for the outside parser. If the resulting parser from the Chunk-and-Pass pipeline obtained an accuracy-speed combination that pushed the Pareto front, it would be a success. Of course, there could exist a network size

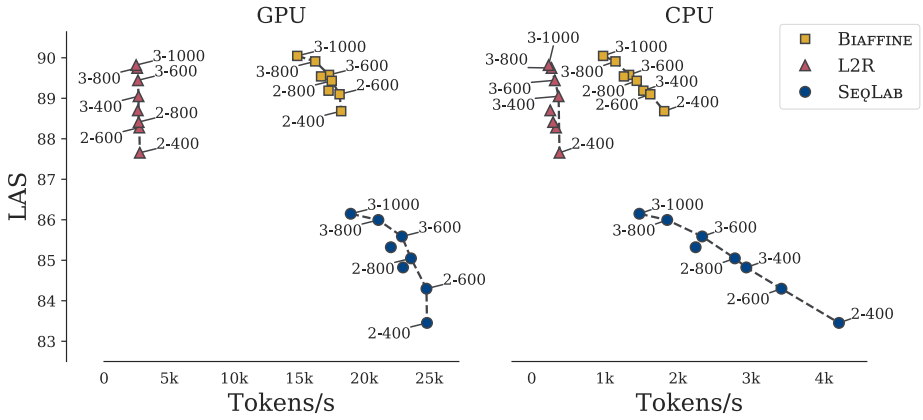


Figure 3.6: Pareto fronts for L2R, Biaffine, and SeqLab on the dev set.

for the outside parsers that maximises the accuracy on the compressed data and it might not be the largest. However, already this process requires many models to be trained with variations across different axes, so we select this network size as it seems sensible.

Setting NPMI thresholds Figure 3.7 shows the compression rate against NPMI for the different treebanks and Table 3.14 shows the NPMI thresholds used to obtain compression rates of 1.2, 1.4, and 1.6. Different values of NPMI are needed for each treebank to obtain the same compression rate. We decided to compare the same compression rates rather than NPMI thresholds as it is much more intuitive to compare.

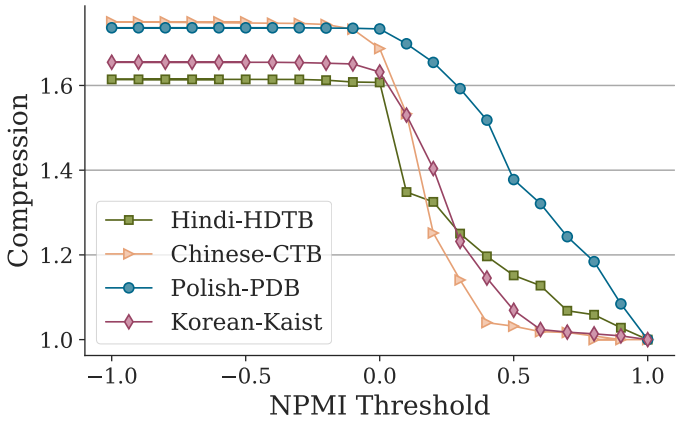


Figure 3.7: Compression ratio against NPMI threshold for training and development data for treebanks used in experiments.

	Compression		
	1.2	1.4	1.6
Hindi	0.388 (1.198)	0.001 (1.354)	0.000 (1.607)
Korean	0.335 (1.209)	0.203 (1.399)	0.001 (1.575)
Polish	0.784 (1.201)	0.467 (1.401)	0.275 (1.600)
Chinese	0.235 (1.204)	0.137 (1.429)	0.053 (1.606)

Table 3.14: NPMI thresholds with exact compression value in parenthesis for both train an dev.

3.4.6 Results

Figure 3.8 shows the average accuracy (averaged across treebanks and across the two tasks of predicted chunk tags and relation labels) for chunkers with different network sizes. The first

thing that stands out is that the speed of the chunkers on GPU is comparable to the standard baseline OUTSIDE parsers (the SeqLab is pretty much the same setup as the chunker where the encoding labels are replaced with chunk tags). Based on this, the CHUNK-AND-PASS system cannot improve parsing efficiency on GPU. However, they are faster on CPU, but in order for the whole pipeline to be faster, the time the OUTSIDE parsers take to process the compressed data needs to be short.

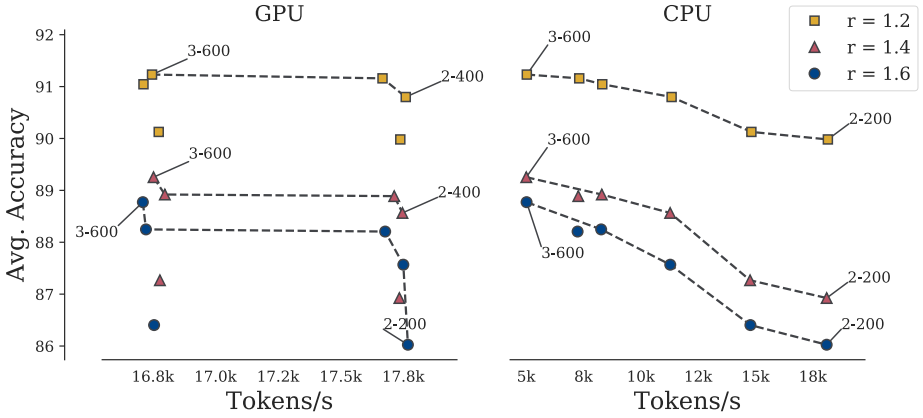


Figure 3.8: Pareto front for different chunker systems. Inference speed is calculated using inference time for chunkers and outside parsers and the time take to collate both output files into a final predicted treebank.

It is clear that the more we compress the data, the worse the chunker performance is. Based on the performance of the chunker networks, we opted to train full pipelines using the chunker with 3 BiLSTM layers and 600 nodes and also the chunker with 2 layers and 200 nodes, effectively the most accurate and the fastest systems on the Pareto front. Table 3.15 shows the accuracy values for these two systems, including LAS for the nodes predicted by the chunker. The performance is quite high across the board, but there is a clear distinction between the two chunkers which is clearer for the higher compression rate, i.e. the average LAS for the smaller network is 84.56 and is 87.48 for the larger network.

		1.2			1.4			1.6		
		$F1_C$	Acc_{rel}	LAS_i	$F1_C$	Acc_{rel}	LAS_i	$F1_C$	Acc_{rel}	LAS_i
2-200	Chinese	87.83	88.84	87.60	82.61	84.33	84.36	81.09	83.71	82.64
	Hindi	90.97	92.19	84.10	91.99	92.46	81.61	90.79	92.33	86.01
	Korean	84.68	85.78	83.94	80.46	83.75	84.53	81.28	85.49	83.01
	Polish	94.78	94.78	93.17	89.74	90.06	88.08	85.84	87.67	86.58
	Average	89.56	90.40	87.20	86.20	87.65	84.64	84.75	87.30	84.56
3-600	Chinese	89.78	90.64	88.67	85.77	87.40	86.17	84.79	87.04	85.99
	Hindi	92.56	93.25	86.15	93.93	93.94	85.59	92.73	93.99	89.39
	Korean	85.75	87.08	85.42	83.43	85.90	86.08	84.34	87.51	85.26
	Polish	95.32	95.47	94.53	91.59	92.07	90.10	89.11	90.67	89.29
	Average	90.85	91.61	88.69	88.68	89.83	86.98	87.74	89.80	87.48

Table 3.15: Chunker performance for model structures used for full system for different compression values.

Table 3.16 shows the full performance of the CHUNK-AND-PASS pipeline for each treebank and each OUTSIDE parser for both the dev and test data. The drop in LAS is severe on the development data at about 9 or 10 points for each OUTSIDE parser, with the worst drops coming for Korean and Chinese. The drop in accuracy is catastrophic on the test data where the decrease is about 20 points for all OUTSIDE parsers. We note that issue appears to be with the OUTSIDE parsers and the decrease in contextual information as the chunkers' performance is fairly high. It appears that the chunk embeddings and the full characters of

each chunk aren't enough to offset this.

Chunker		Biaffine					SeqLab					L2R				
Size	r	zh	hi	ko	pl	avg	zh	hi	ko	pl	avg	zh	hi	ko	pl	avg
2-200	1.2	75.90	86.53	77.22	85.10	81.19	69.60	83.83	72.50	81.12	76.76	75.25	87.29	76.72	83.65	80.73
	1.4	71.65	84.17	74.20	82.24	78.06	66.72	82.22	70.30	77.97	74.30	71.31	84.37	72.61	80.29	77.15
	1.6	69.97	84.05	71.65	78.88	76.14	65.26	81.58	69.08	74.88	72.70	69.67	83.72	70.95	77.20	75.38
3-600	1.2	76.36	87.19	77.56	85.97	81.77	71.14	85.38	73.12	81.87	77.88	76.17	87.70	76.83	84.49	81.30
	1.4	73.94	85.88	74.64	83.65	79.53	69.70	84.19	71.16	79.72	76.19	73.57	86.22	73.67	82.50	78.99
	1.6	72.81	86.04	72.86	81.36	78.27	69.13	84.22	71.20	77.84	75.60	72.30	85.63	72.75	80.27	77.74
Baseline		82.98	91.16	86.03	88.64	87.20	78.41	89.31	81.80	86.64	84.04	83.41	91.20	85.28	88.84	87.18

(a) **Dev LAS**

Chunker		Biaffine					SeqLab					L2R				
Size	r	zh	hi	ko	pl	avg	zh	hi	ko	pl	avg	zh	hi	ko	pl	avg
2-200	1.2	71.72	77.51	60.08	73.54	70.71	63.54	77.44	52.77	70.76	66.13	71.73	85.04	61.00	69.79	71.89
	1.4	64.53	76.37	60.71	71.24	68.21	59.52	72.98	55.12	67.30	63.73	66.47	80.40	56.07	66.42	67.34
	1.6	62.09	76.75	59.99	69.07	66.97	55.91	70.75	54.43	63.87	61.24	63.14	79.55	57.18	63.12	65.75
3-600	1.2	67.66	77.42	57.88	74.50	69.36	63.98	81.09	56.11	70.69	67.97	70.18	85.20	61.02	71.23	71.91
	1.4	65.24	74.29	59.41	72.39	67.83	61.08	76.94	54.40	68.19	65.15	66.10	82.42	57.83	68.57	68.73
	1.6	63.87	77.70	60.07	71.52	68.29	59.68	74.52	56.38	66.13	64.18	66.35	78.87	59.06	65.57	67.46
Baseline		83.47	90.94	85.56	88.86	87.21	78.91	89.26	81.68	87.20	84.26	83.65	91.18	84.47	89.34	87.16

(b) **Test LAS**

Table 3.16: Full results on dev and test data.

As discussed above and shown in Table 3.12, OOV is a major issue here and is more pronounced for the test data, as we developed the chunk vocabulary space with the development data to try and make it as wide as possible. So we evaluated the chunk-and-pass pipeline using the OOV procedure with the larger chunker network to see if it would offset some of the drop in accuracy. Table 3.17 shows the results for this experiment. For the L2R parser there is a notable increase in accuracy, but it only dents the deficit compared to the baseline performance. For the other two parsers, there is little (or no) increase for each treebank. This suggests that there might be some issue with how these systems are utilising the input for the chunk embedding form. However, even if this is true, the moderate increase in accuracy seen for the L2R parser wouldn't salvage these either.

		LAS				
		zh	hi	ko	pl	avg
Biaffine	CNP	63.87	77.70	60.07	71.52	68.29
	w/ OOV	63.90	77.70	60.07	72.23	68.47
Seq-Lab	CNP	59.68	74.52	56.38	66.13	64.18
	w/ OOV	59.68	74.52	56.38	66.31	64.22
L2R	CNP	66.35	78.87	59.06	65.57	67.46
	w/ OOV	71.72	83.66	64.80	72.92	73.28

Table 3.17: Impact of using OOV handling procedure for chunker network size of 3-600 with compression r= 1.6 on test data.

We initially intended to use pretrained word embeddings for the chunkers too, but due to a bug in the code we ended up using randomly initialised embeddings. So we trained chunkers with external embeddings to see if we could increase their accuracy, even though we consider the main issue of the chunk-and-pass pipeline to be a lack of context for the outside parsers. Tables 3.18 and 3.19 show the chunker label accuracies and the relation accuracies for the chunkers with and without pretrained embeddings. The performance actually decreases when using them for most treebanks.

Chunker Size	r	Chinese w/ ext	Chinese w/o	Hindi w/ ext	Hindi w/o	Korean w/ ext	Korean w/o	Polish w/ ext	Polish w/o
2-200	1.2	87.20	87.95	90.66	91.58	85.34	86.11	94.30	94.86
	1.4	81.48	82.59	91.00	92.00	79.47	80.45	89.12	89.76
	1.6	79.78	81.10	89.12	90.79	80.56	81.32	84.66	85.81
3-600	1.2	89.14	89.90	92.50	93.18	87.44	87.40	95.31	95.39
	1.4	85.01	85.76	93.18	93.94	82.97	83.47	91.35	91.65
	1.6	83.93	84.77	91.94	92.73	83.21	84.40	88.52	89.10

Table 3.18: Testing impact of external embeddings on chunk accuracy.

Chunker Size	r	Chinese w/ ext	Chinese w/o	Hindi w/ ext	Hindi w/o	Korean w/ ext	Korean w/o	Polish w/ ext	Polish w/o
2-200	1.2	88.30	88.96	91.90	92.79	86.87	87.32	94.33	94.88
	1.4	83.33	84.32	91.57	92.46	83.01	83.73	89.61	90.06
	1.6	82.55	83.74	91.48	92.33	84.99	85.57	86.90	87.66
3-600	1.2	90.01	90.74	93.37	93.90	88.73	88.80	95.36	95.54
	1.4	86.52	87.39	93.21	93.94	85.72	85.90	91.84	92.13
	1.6	86.24	87.06	93.36	94.01	86.90	87.60	90.26	90.66

Table 3.19: Testing impact of external embeddings on chunker's relation accuracy.

Clearly the concept has failed to deliver. In Figure 3.9, the original Pareto front across the baseline OUTSIDE parsers (from Figure 3.6) is shown with the CHUNK-AND-PARSER systems. Not only are these parsing systems considerably less accurate, they're also slower. Granted if we had use a smaller network for the OUTSIDE parsers, they would have been faster. But how low would the accuracy have been? It's hard to say for sure because a smaller network *could* potentially be better suited for processing the compressed data. But it would have to be considerably better suited to offset the huge drop in accuracy seen for the models trained here.

Before we were aware of just how bad these parsing systems were, we considered it important to measure the training costs compared to the baseline models. This is another trade-off in that training a multi-component system typically takes longer and therefore consumes more energy. So if the CHUNK-AND-PASS parser had been more efficient at inference time, would that have come at too high a cost with respect to training cost? Table 3.20 shows the energy consumed trained each CHUNK-AND-PASS system and the equivalent baseline. The training costs are actually often lower for the CHUNK-AND-PASS systems than the simple baselines. This is completely dependent on training time as no system is clearly more energy greedy than any other based on these results.

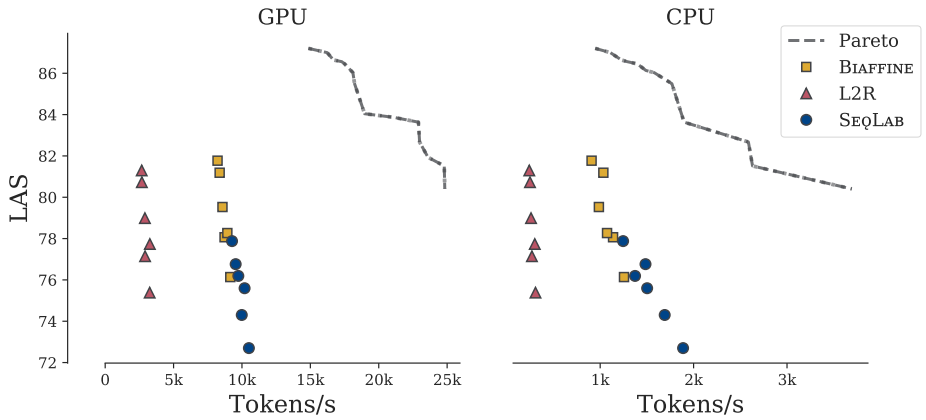


Figure 3.9: Pareto front for different chunk and pass systems on dev set.

Chunker Size	r	Total Energy (MJ) Biaffine	SeqLab	L2R	Total Time (hours) Biaffine	SeqLab	L2R
2-200	1.2	0.85	1.17	0.62	11.7	13.8	7.1
	1.4	1.19	1.08	0.61	13.2	12.3	6.9
	1.6	0.94	0.93	0.66	10.7	10.4	7.3
3-600	1.2	1.01	1.13	0.76	11.1	13.2	8.9
	1.4	0.98	1.11	0.77	10.8	12.6	8.9
	1.6	1.04	0.96	0.70	11.1	10.8	7.8
Baseline		1.01	1.07	0.92	11.2	11.9	11.1

Table 3.20: Total energy consumed during training and total training time for each parser system for different chunkers across all 4 treebanks used. Values include training costs from training chunkers, except for baseline where no chunker is used (3-1000 shown).

3.4.7 Summary

We have introduced chunk-and-pass parsing and evaluated its efficacy. It can only really be described as an abject failure with respect to the underlying goal. However, the NPMI labelling system could potentially be useful to generate features for other systems based on the results in Section 3.3 and is infinitely faster than using an evolutionary technique. Beyond that, we had to evaluate three leading parsing systems in a thorough way and in a consistent way. This has resulted in a very clear picture of modern parsers: Biaffine parsers are really the default choice unless you have very restrictive time limits (and are restricted to CPUs) then the sequence labelling systems are a viable choice.

3.5 Conclusion

In this chapter, we evaluated chunking and its interplay with parsing. In Section 3.3, we used an evolutionary algorithm to pick the best set of rules used to extract chunks for treebanks and evaluated their impact on POS tagging, morphological feature tagging, and dependency parsing. These automatically extracted chunks were seen to be broadly helpful for these tasks, especially when utilised in a MTL setup. We then introduced a way of extracting chunks using information theory in Section 3.4. We use NPMI to rank potential rules based on the POS tags of tokens, the POS tag of their head, and the relation connecting them. We then used these to develop chunk-and-pass parsers. These parsers were an abject failure. But the NPMI method of extracting chunks developed for them could be useful in a similar vein as those extracted using the evolutionary algorithm and has the added benefit of not taking days to run.

Chapter 4

Neural Network Distillation

Work in this chapter is based on published work in Anderson and Gómez-Rodríguez (2020a) and Dehouck et al. (2020).

In this chapter we present our attempts to develop more efficient dependency parsers using *teacher-student* distillation, where a larger pre-trained network is used to guide the training of a smaller network. This method is described in detail in Section 4.3. The main goal is to develop smaller, more efficient parsers while maintaining as much accuracy as possible. In Section 4.4, we use distillation to develop fast and accurate parsers and offer a full analysis of these parsers with respect to inference performance. In Section 4.5, we detail our contribution to the IWPT 2020 shared task where distilled dependency parsers formed the basis of our system. It also presents other techniques used in conjunction with distillation to increase efficiency and offers more details about the training cost associated with distilling parsers.

4.1 Introduction

Latterly, the environmental impact of AI and NLP's dependency on deep neural networks has come under scrutiny (Schwartz et al., 2019; Strubell et al., 2019). This has coincided with a renewed push for efficiency in NLP so as to make systems more suitable for different contexts, be it in hardware impaired conditions, large web-scale applications, or a host of other considerations (Strzyz et al., 2019b; Clark et al., 2019b; Vilares et al., 2019; Junczys-Dowmunt et al., 2018). Beyond developing systems to be greener, increasing the efficiency of models makes them more cost-effective, which is a compelling argument even for people who might downplay the extent of anthropogenic climate change.

In conjunction with this push for greener AI, NLP practitioners have turned to the problem of developing models that are not only accurate but also efficient, so as to make them more readily deployable across different machines with varying computational capabilities (Strzyz et al., 2019b; Clark et al., 2019b; Vilares et al., 2019; Junczys-Dowmunt et al., 2018). This is in contrast with the recently popular principle of *make it bigger, make it better* (Devlin et al., 2019; Radford et al., 2019).

Here we explore *teacher-student* distillation as a means of increasing the efficiency of neural network systems used to undertake dependency parsing. To do so, we start with the biaffine parser. The biaffine parser is not only one of the most accurate parsers, it is the fastest implementation by almost an order of magnitude among leading performing parsers.

In the following section (Section 4.2) we offer a brief overview of related work in the space

of model compression. In Section 4.3, we give details on *teacher-student* distillation and how it was implemented for this work. Section 4.4 details the methodology and corresponding results obtained in the original work and Section 4.5 discusses our contribution to the IWPT 2020 shared task on Enhanced Universal Dependency parsing where we used distillation and other techniques to focus on efficiency.

4.2 Related work

Model compression has been under consideration for almost as long as neural networks have been utilised, e.g. LeCun et al. (1990) introduced a pruning technique which removed weights based on a locally predicted contribution from each weight so as to minimise the perturbation to the error function. More recently, Han et al. (2015) introduced a means of pruning a network up to 40 times smaller with minimal effect on performance. Hagiwara (1994) and Wan et al. (2009) utilised magnitude-based pruning to increase network generalisation. More specific to NLP, See et al. (2016) used absolute-magnitude pruning to compress neural machine translation systems by 40% with minimal loss in performance. However, pruning networks leaves them in an irregularly sparse state which cannot be trivially re-cast into less sparse architectures. Sparse tensors could be used for network layers to obtain real-life decreases in computational complexity, however, current deep learning libraries lack this feature. Anwar et al. (2017) introduced structured pruning to account for this, but this kernel-based technique is restricted to convolutional networks. More recently Voita et al. (2019) pruned the heads of the attention mechanism in their neural machine translation system and found that the remaining heads were linguistically salient with respect to syntax, suggesting that pruning could also be used to undertake more interesting analyses beyond merely compressing models and helping generalisation.

Ba and Caruana (2014) and Hinton et al. (2015) developed distillation as a means of network compression from the work of Bucilă et al. (2006), who compressed a large ensemble of networks into one smaller network. Similar and more recent work used this method of compressing many models into one to achieve high parsing performance (Kuncoro et al., 2016). *Teacher-student* distillation is the process of taking a large network, the *teacher*, and transferring its knowledge to a smaller network, the *student*. *Teacher-student* distillation has successfully been exploited in NLP for machine translation, language modelling, and speech recognition (Kim and Rush, 2016; Yu et al., 2018; Lu et al., 2017). Beyond that it has also been successfully used in conjunction with exploring structured linguistic prediction spaces (Liu et al., 2018). Latterly, it has also been used to distill task-specific knowledge from BERT (Tang et al., 2019).

Other compression techniques have been used such as low-rank approximation decomposition (Yu et al., 2017), vector quantisation (Wu et al., 2016), and Huffman coding (Han et al., 2016). For a more thorough survey of current neural network compression methods see Cheng et al. (2018).

4.2.1 Current parser performance

Table 4.1 shows performance details of current leading dependency parsers on the English Penn Treebank (PTB) with predicted POS tags from the Stanford POS tagger (Marcus and Marcinkiewicz, 1993; Toutanova et al., 2003). The biaffine parser of Dozat and Manning (2017) offers the best trade-off between accuracy and parsing speed with the HPSG parser of Zhou and Zhao (2019) achieving the absolute best reported accuracy but with a reported parsing speed of roughly one third of the biaffine's parsing speed. It is important to note that direct comparisons between systems with respect to parsing speed are wrought with compounding variables, e.g. different GPUs or CPUs used, different number of CPU cores, different batch sizes, and often hardware is not even reported.

We therefore run a subset of parsers locally to achieve speed measurements in a controlled

	speed (sent/s) GPU	CPU	UAS	LAS
Pointer-TD (Ma et al., 2018)	-	10.2†	95.87†	94.19†
Pointer-LR (Fernández-González and Gómez-Rodríguez, 2019)	-	23.1†	96.04†	94.43†
HPSG (Zhou and Zhao, 2019)	158.7†	-	96.09†	94.68†
BIST - Transition (Kiperwasser and Goldberg, 2016)	-	76±1‡	93.9†	91.9†
BIST - Graph (Kiperwasser and Goldberg, 2016)	-	80±0‡	93.1†	91.0†
Biaffine (Dozat and Manning, 2017)	411†	-	95.74†	94.08†
CM (Chen and Manning, 2014)	-	654†	91.80†	89.60†
SeqLab (Strzyz et al., 2019b)	648±20‡	101±2‡	93.67‡	91.72‡
UUParser (Smith et al., 2018)	-	42±1	94.63	92.77
Biaffine (PyTorch)	1003±3	53±0	95.74	94.07
SeqLab	1064±13	99±1	93.46	91.49
Biaffine-D20	1189±4	391±2	92.84	90.73
Biaffine-D40	**1153±3**	**96±0**	**94.59**	**92.64**
Biaffine-D60	1112±6	71±1	94.78	92.86
Biaffine-D80	1033±5	61±0	94.84	92.95

Table 4.1: Speed and accuracy performance for leading parsers and parsers from our distillation method, Biaffine-Dπ compressing to π% of the original model, for the English PTB with POS tags predicted from the Stanford POS tagger. In the first table block, † denotes values taken from the original paper and ‡ from Strzyz et al. (2019b). Values with no superscript (corresponding to the models in the second and third blocks) are from running the models on our system locally with a single CPU core for both CPU and GPU speeds (averaged over 5 runs) and with a batch size of 256 (excluding UUParser which doesn't support batching) with GloVe 100 dimension embeddings.

environment, also shown in Table 4.1. We compare a PyTorch implementation of the biaffine parser (which runs more than twice as fast as the reported speed of the original implementation); the UUParser from Smith et al. (2018) which is one of the leading parsers for Universal Dependency (UD) parsing; a sequence-labelling dependency parser from Strzyz et al. (2019b) which has the fastest reported parsing speed amongst modern parsers; and also distilled biaffine parsers from our implementation described below. All speeds measured here are with the system run with a single CPU core for both GPU and CPU runs.[1]

4.3 Teacher-student distillation

The essence of model distillation is to train a model and subsequently use the patterns it learnt to influence the training of a smaller model. For *teacher-student* distillation, the smaller model, the *student*, explicitly uses the information learnt by the larger original model, the *teacher*, by comparing the distribution of each model's output layer. We use the Kullback-Leibler divergence to calculate the loss between the teacher and the student:

$$\mathcal{L}_{KL} = -\sum_{t \in b} \sum_{i} P(\mathbf{x}_i) \log \frac{P(\mathbf{x}_i)}{Q(\mathbf{x}_i)} \tag{4.1}$$

where P is the probability distribution from the teacher's softmax layer, Q is the probability distribution from the student's, and $\mathbf{x}_i$ is an input vector to the softmax corresponding to token w_i of a given tree t for all trees in batch b.

For our implementation, there are two probability distributions for each model, one for the arc prediction and one for the label prediction. By using the distributions of the teacher rather than just using the predicted arc and label, the student can learn more comprehensively about which arcs and labels are very unlikely in a given context, i.e. if the teacher makes a mistake in its prediction, the distribution might still carry useful information such as having

[1] This is for ease of comparability. Parsing can trivially be parallelised by allocating sentences to different cores, so speed per core is an informative metric to compare parsers (Hall et al., 2014).

a similar probability for y_g and y_p which can help guide the student better rather than just learning to copy the teacher's predictions.

In addition to the loss with respect to the teacher's distributions, the student model is also trained using the loss on the gold labels in the training data. We use categorical cross entropy to calculate the loss on the student's predicted head classifications:

$$\mathcal{L}_{CE} = -\sum_{t \in b} \sum_{i} \log p(h_i | \mathbf{x}_i) \tag{4.2}$$

where h_i is the true head position for token w_i, corresponding to the softmax layer input vector $\mathbf{x}_i$, of tree t in batch b. Similarly, categorical cross entropy is used to calculate the loss on the predicted arc labels for the student model. The total loss for the student model is therefore:

$$\begin{aligned} \mathcal{L} = \mathcal{L}_{KL}(T_h, S_h) + \mathcal{L}_{KL}(T_{lab}, S_{lab}) \\ + \mathcal{L}_{CE}(h) + \mathcal{L}_{CE}(lab) \end{aligned} \tag{4.3}$$

where $\mathcal{L}_{CE}(h)$ is the loss for the student's predicted head positions, $\mathcal{L}_{CE}(lab)$ is the loss for the student's predicted arc label, $\mathcal{L}_{KL}(T_h, S_h)$ is the loss between the teacher's probability distribution for arc predictions and that of the student, and $\mathcal{L}_{KL}(T_{lab}, S_{lab})$ is the loss between label distributions. This combination of losses broadly follows the methods used in Tang et al. (2019) but is altered to fit the biaffine parser.

4.4 Experiment 1

Here we give details of the original experiment we undertook to evaluate distillation for neural dependency parsers. We find that distillation maintains accuracy performance close to that of the full model and obtains far better accuracy than simply implementing equivalent model size reductions by changing the parser's network size and training normally. We can compress a parser to 20% of its trainable parameters with minimal loss in accuracy and with a speed 2.30x (1.19x) faster than that of the original model on CPU (GPU).

4.4.1 Methodology

We train biaffine parsers and apply the *teacher-student* distillation method to compress these models into a number of different sizes for a number of Universal Treebanks v2.4 (UD) (Nivre et al., 2019). We use the hyperparameters from Dozat and Manning (2017), but use a PyTorch implementation for our experiments which obtains the same parsing results and runs faster than the reported speed of the original (see Table 4.1).[2] The hyperparameter values can be seen in Table 4.2. During distillation dropout is not used as in earlier experiments with dropout performance was hampered. And the subsequent work on distillation (4.5) which uses dropout also didn't perform as well, but it isn't clear if this is the cause of the poorer performance, e.g. different treebanks were used, UPOS tags weren't, and no pre-trained embeddings were used. Beyond lexical features, the model only utilises universal part-of-speech (UPOS) tags. Gold UPOS tags were used for training and at runtime. Also, we used gold sentence segmentation and tokenisation. We opted to use these settings to compare models under homogeneous settings, so as to make reproducibility of and comparability with our results easier. In hindsight, it might have been better to have used predicted tags (as we did for the PTB results).

[2]The implementation can be found at `github.com/zysite/biaffine-parser`. Beyond adding our distillation method, we also included the Chu-Liu/Edmonds' algorithm, as used in the original, to enforce well-formed trees.

hyperparameter	value
word embedding dimensions	100
pos embedding dimensions	100
embedding dropout	0.33
BiLSTM dimensions	400
BiLSTM layers	3
arc MLP dimensions	500
label MLP dimensions	100
MLP layers	1
learning rate	0.2
dropout	0.33
momentum	0.9
L2 norm λ	0.9
annealing	$0.75^{\wedge}(t/5000)$
ϵ	1×10^{-12}
optimiser	Adam
loss function	cross entropy
epochs	100

Table 4.2: Hyperparameters for full-sized baseline models.

Data We use the subset of UD treebanks suggested by de Lhoneux et al. (2017c) from v2.4, so as to cover a wide range of linguistic features, linguistic typologies, and different dataset sizes. We make some changes as this set of treebanks was chosen from a previous UD version. We exchange Kazakh with Uyghur because the Kazakh data does not include a development set and Uyghur is a closely related language. We also exchange Ancient-Greek-Proiel for Ancient-Greek-Perseus because it contains more non-projective arcs as this was the original justification for including Ancient Greek. Further, we follow Smith et al. (2018) and exchange Czech-PDT with Russian-GSD. We also included Wolof as African languages were wholly unrepresented in the original collection of suggested treebanks (Dione, 2019). Details of the treebanks pertinent to parsing can be seen in Table 4.3. We use pretrained word embeddings from FastText (Grave et al., 2018) for all but Ancient Greek, for which we used embeddings from Ginter et al. (2017), and Wolof, for which we used embeddings from Heinzerling and Strube (2018). When necessary, we used the algorithm of Raunak (2017) to reduce the embeddings to 100 dimensions.

For each treebank we then acquired the following models:

i **Baseline 1**: Full-sized model is trained as normal and undergoes no compression technique.

ii **Baseline 2**: Model is trained as normal but with equivalent sizes of the distilled models (20%, 40%, 60%, and 80% of the original size) and undergoes no compression technique. These models have the same overall structure of baseline 1, with just the number of dimensions of each layer changed to result in a specific percentage of trainable parameters of the full model.

iii **Distilled**: Model is distilled using the *teacher-student* method. We have four models where the first is distilled into a smaller network with 20% of the parameters of the original, the second 40%, the third 60%, and the last 80%. The network structure and parameters of the distilled models are the exact same as those of the baseline 2 models.

Hardware For evaluating the speed of each model when parsing the test sets of each treebank we set the number of CPU cores to be one and either ran the parser using that solitary core or using a GPU (using a single CPU core too). The CPU used was an Intel Core i7-7700 and the GPU was an Nvidia GeForce GTX 1080.[3]

[3]Using Python 3.7.0, PyTorch 1.0.0, and CUDA 8.0.

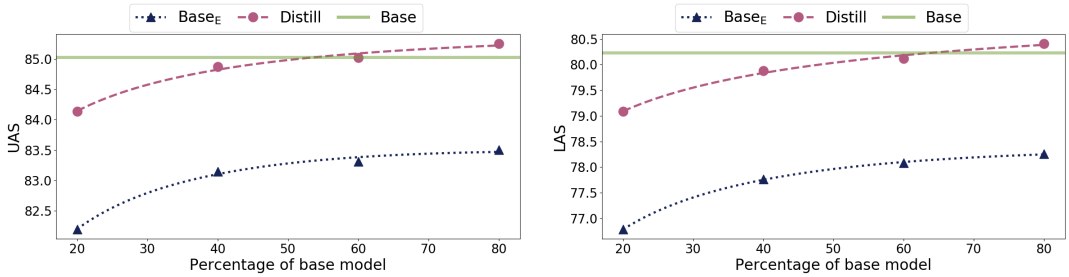
	number of trees			avg sent. length			avg. arc length			non-proj. arc pct		
	train	dev	test	train	dev	test	train	dev	test	train	dev	test
Ancient-Greek-Pers.	11476	1137	1306	14.9	20.5	17.0	4.1	4.5	4.1	23.9	23.2	23.5
Chinese-GSD	3997	500	500	25.7	26.3	25.0	4.7	4.9	4.7	0.1	0.0	0.3
English-EWT	12543	2002	2077	17.3	13.6	13.1	3.7	3.5	3.6	1.0	0.6	0.6
Finnish-TDT	12217	1364	1555	14.3	14.4	14.5	3.4	3.4	3.4	1.6	1.9	1.8
Hebrew-HTB	5241	484	491	27.3	24.6	26.0	3.9	3.8	3.7	0.8	0.8	0.9
Russian-GSD	3850	579	601	20.5	21.2	19.9	3.5	3.7	3.7	1.1	1.0	1.2
Tamil-TTB	400	80	120	16.8	16.8	17.6	3.5	3.7	3.7	0.3	0.0	0.2
Uyghur-UDT	1656	900	900	12.6	12.8	12.5	3.5	3.5	3.5	1.1	1.3	1.4
Wolof-WTB	1188	449	470	20.8	23.9	23.1	3.5	3.8	3.6	0.4	0.4	0.5

Table 4.3: Statistics for salient features with respect to parsing difficulty for each UD treebank used: number of trees, the number of data instances; average sent length, the length of each data instance on average; average arc length, the mean distance between heads and dependents; non.proj. arc pct, the percentage of non-projective arcs in a treebank.

Experiment We compare the performance of each model on the aforementioned UD treebanks with respect to both UAS and LAS. We also evaluate the differences in inference time for each model on CPU and GPU with respect to sentences per second and tokens per second. We report sentences per second as this has been the measurement traditionally used in most of the literature, but we also use tokens per second as this more readily captures the difference in speed across parsers for different treebanks where the sentence length varies considerably. We also report the number of trainable parameters of each distilled model and how they compare to the baseline, as this is considered a good measure of how green a model is in lieu of the number of floating point operations (FPO) (Schwartz et al., 2019).[4]

4.4.2 Results and Discussion

Figure 4.1 shows the average attachment scores across all test treebanks (all results presented in this section are on the test treebanks) for the distilled models and the equivalent-sized base models against the size of the model relative to the original full model. There is a clear gap in performance between these two sets of models with roughly 2 points of UAS and LAS more for the distilled models. This shows that the distilled models do actually manage to leverage the information from the original full model. The full model's scores are also shown and it is clear that on average the model can be distilled to 60% with no loss in performance. When compressing to 20% of the full model, the performance only decreases by about 1 point for both UAS and LAS.



(a) (b)

Figure 4.1: UAS (a) and LAS (a) against the model size relative to the original full-sized model: Base$_E$, the baseline models of equivalent size to the distilled models; Distill, the distilled models; Base, the performance of the original full-sized model.

Figures 4.2a and 4.2b show the differences in UAS and LAS for the models distilled to

[4]There exist a number of packages for computing the FPO of a model but, to our knowledge, as of yet they do not include the capability of dealing with LSTMs.

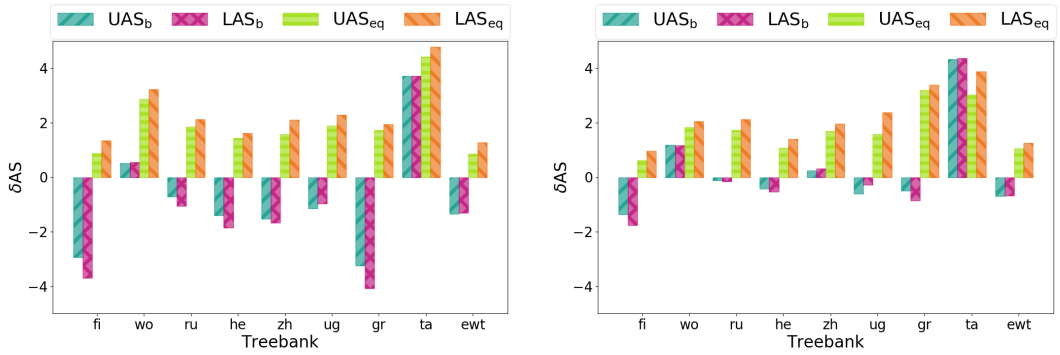


Figure 4.2: Delta UAS and LAS for when comparing both the original base model and equivalent-sized base models for each treebank for two of our distilled models: (a) D-20, 20% of original model and (b) D-80, 80% of original model.

20% and 80% respectively for each treebank when compared to the equivalent sized baseline model and the full baseline model. The distilled models far outperform the equivalent-sized baselines for all treebanks. It is clear that for the smaller model some treebanks suffer more when compressed to 20% than others when compared to the full baseline model, e.g. Finnish-TDT and Ancient-Greek-Perseus. These two treebanks have the largest percentage of non-projective arcs (as can be seen in Table 4.3) which could account for the decrease in performance, with a more powerful model required to account for this added syntactic complexity.

	gr	**zh**	**en**	**fi**	**he**	**ru**	**ta**	**ug**	**wo**	**avg**
Full	75.5	88.2	90.8	90.5	90.8	88.9	76.9	75.2	88.5	85.0
B-20	70.5	85.1	88.6	86.7	87.9	86.3	76.2	72.2	86.1	82.2
B-40	72.2	86.1	88.9	87.7	88.5	87.1	78.4	73.0	86.5	83.2
B-60	72.0	86.7	89.5	88.1	88.7	87.1	77.5	72.7	87.5	83.3
B-80	71.8	86.7	89.1	88.5	89.3	87.1	78.2	73.0	87.8	83.5
D-20	72.3	86.7	89.5	87.6	89.4	88.2	80.6	74.1	89.0	84.1
D-40	74.0	87.9	89.9	89.5	89.4	88.4	80.9	74.5	89.4	84.9
D-60	74.2	88.3	90.1	89.4	90.0	88.6	80.4	74.5	89.5	85.0
D-80	75.0	88.4	90.1	89.2	90.3	88.8	81.2	74.6	89.6	85.3

Table 4.4: Full UAS values for each model and for each test treebank where Full means the original sized model, B-X means training a model with X% of the trainable parameters of the original model, and D-X means distilling to a model with X% of the trainable parameters of the original model.

	gr	**zh**	**en**	**fi**	**he**	**ru**	**ta**	**ug**	**wo**	**avg**
Full	70.4	85.9	89.0	88.6	88.6	85.2	71.0	58.9	84.5	80.2
B-20	64.4	82.1	86.4	83.6	85.1	82.0	69.9	55.6	81.8	76.8
B-40	66.4	83.5	86.8	84.8	85.6	83.1	71.8	55.7	82.2	77.8
B-60	66.4	84.0	87.5	85.5	86.3	83.1	70.9	55.9	83.1	78.1
B-80	66.2	84.3	87.1	85.9	86.6	82.9	71.5	56.2	83.6	78.3
D-20	66.4	84.2	87.7	84.9	86.7	84.2	74.7	57.9	85.0	79.1
D-40	68.3	85.6	88.0	86.9	87.0	84.6	74.7	58.3	85.5	79.9
D-60	68.7	85.9	88.3	87.1	87.5	84.7	74.5	58.6	85.8	80.1
D-80	69.6	86.2	88.3	86.9	88.0	85.0	75.4	58.6	85.7	80.4

Table 4.5: Full LAS values for each model and for each test treebank where Full means the original sized model, B-X means training a model with X% of the trainable parameters of the original model, and D-X means distilling to a model with X% of the trainable parameters of the original model.

However, the two smallest treebanks, Tamil-TTB and Wolof-WTB, actually increase in

accuracy when using distillation, especially Tamil-TTB, which is by far the smallest treebank, with an increase in UAS and LAS of about 4 points over the full base model. This is likely the result of over-fitting when using the larger, more powerful model, so that reducing the model size actually helps with generalisation.

These observations are echoed in the results for the model distilled to 80%, where most treebanks lose less than a point for UAS and LAS against the full baseline, but have a smaller increase in performance over the equivalent-sized baseline. This makes sense as the model is still close in size to the full baseline and still similarly powerful. The increase in performance for Tamil-TTB and Wolof-WTB are greater for this distilled model, which suggests the full model doesn't need to be compressed to such a small model to help with generalisation. The full set of attachment scores from our experiments can be seen in Tables 4.4 and 4.5.

	gr	**zh**	**en**	**fi**	**he**	**ru**	**ta**	**ug**	**wo**
Full	12.28	11.98	12.23	12.77	12.04	11.92	11.22	11.45	11.39
D-20	2.5 (19.7)	2.4 (20.2)	2.4 (19.7)	2.6 (19.7)	2.4 (19.2)	2.4 (19.3)	2.3 (19.6)	2.3 (20.2)	2.3 (19.5)
D-40	4.9 (39.3)	4.8 (39.5)	4.9 (39.3)	5.1 (40.2)	4.8 (40.0)	4.7 (39.5)	4.5 (39.3)	4.6 (40.4)	4.6 (39.8)
D-60	7.4 (59.8)	7.2 (60.5)	7.3 (59.8)	7.7 (59.8)	7.2 (59.2)	7.2 (59.7)	6.7 (59.8)	6.9 (60.5)	6.8 (60.2)
D-80	9.8 (80.3)	9.6 (79.8)	9.8 (79.5)	10.2 (80.3)	9.6 (79.2)	9.52 (79.8)	8.9 (79.5)	9.2 (79.8)	9.1 (80.5)

Table 4.6: Trainable model parameters ($\times 10^6$) with percentage of full model in parentheses, where Full means the original sized model and D-X means distilling to a model with X% of the trainable parameters of the original model.

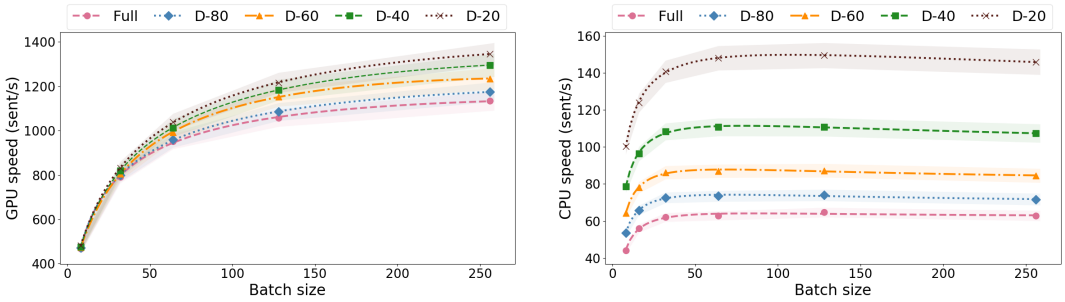
With respect to how green our distilled models are, Table 4.6 shows the number of trainable parameters for each distilled model for each treebank alongside its corresponding full-scale baseline. We report these in lieu of FPO as, to our knowledge, no packages exist to calculate the FPO for neural network layers like LSTMs which are used in our network. These numbers do not depend on the hardware used and strongly correlate with the amount of memory a model consumes. Different algorithms do utilise parameters differently, however, the models compared here are of the same structure and use the same algorithm, so comparisons of the number of trainable model parameters do relate to how much work each respective model does compared to another. Beyond this we offer a nominal analysis of inference energy consumption for each of the model sizes. These measurements can be seen in Table 4.7. The full baseline uses roughly 33% more than the smallest distilled model. This difference is more pronounced when including the energy used to load the models (which might be a consideration if the parser cannot be kept in memory) as the full baseline almost uses twice as much energy as the smallest distilled model.

	Energy (kJ)				
	Full	**D-80**	**D-60**	**D-40**	**D-20**
inference	0.32	0.31	0.27	0.25	0.24
w/ model load	6.91	6.70	6.9	5.95	3.67

Table 4.7: Total inference energy consumption (inference) used for all test treebanks (8K sentences) and also with the energy consumption used to load each of the 9 models (w/ model load). The standard deviation for inference energy consumption was 0.01 exclusively and for the consumption with loading models it ranged from 0.06 to 0.15.

Figures 4.3 and 4.4 show the parsing speeds on CPU and GPU for the distilled models and for the full baseline model in sentences and tokens per second, respectively. The speeds are reported for different batch sizes as this obviously affects the speed at which a neural network can make predictions, but the maximum batch size that can be used on different systems varies significantly. As can be seen in Figures 4.3a and 4.4a, the limiting factor in parsing speed is the bottleneck of loading the data onto the GPU when using a batch size

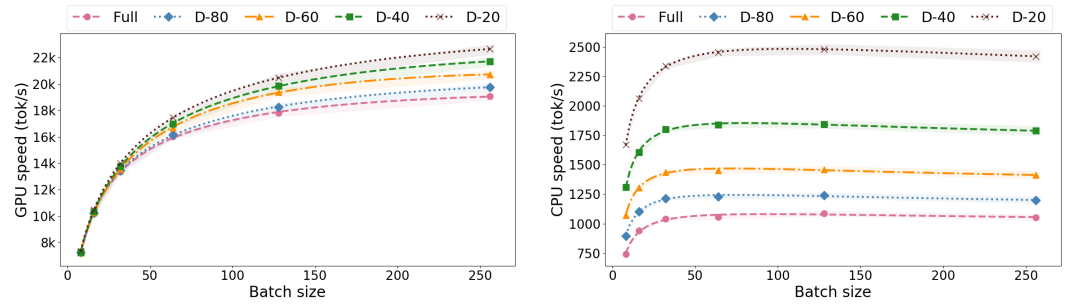
less than ∼50 sentences. However, with a batch size of 256 sentences, we achieve an increase in parsing speed of 19% over the full baseline model when considering tokens per second.



(a) (b)

Figure 4.3: GPU (a) and single core CPU (b) speeds in sentence per second with varying batch sizes for distilled models (D-X) and full-sized base model (Full). Shaded areas show the standard error. Speeds for Tamil-TTB are not included as the test treebank is too small for larger batch sizes.

As expected, a much smaller batch size is required to achieve increases in parsing speed when using a CPU. Even with a batch size of 16 sentences, the smallest model more than doubles the speed of the baseline. For a batch size of 256, the distilled model compressed to 20% increases the speed of the baseline by 130% when considering tokens per second. A full breakdown of the parsing speeds for each treebank and each model when using a batch size of 256 sentences is given in Table 4.8.



(a) (b)

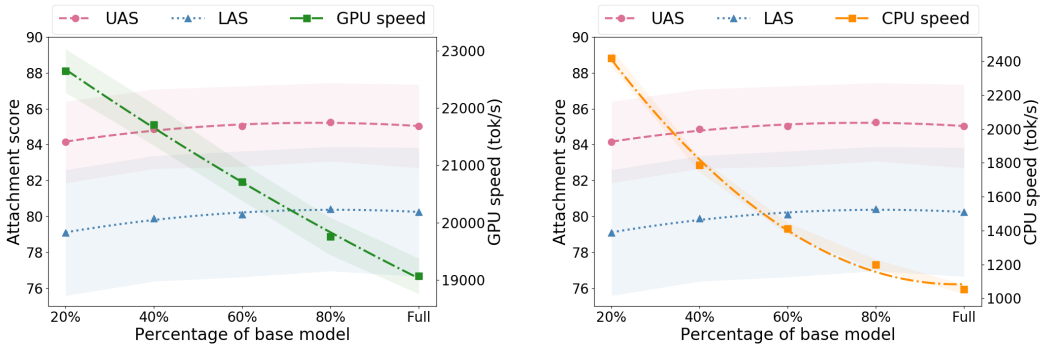
Figure 4.4: GPU (a) and single core CPU (b) speeds in tokens per second with varying batch sizes for distilled models (D-X) and full-sized base model (Full). Shaded areas show the standard error. Speeds for Tamil-TTB are not included as the test treebank is too small for larger batch sizes.

Figure 4.5 shows the attachment scores and the corresponding parsing speed against model size for the distilled model and the full baseline model. These plots clearly show that the cost in accuracy is negligible when compared to the large increase in parsing speed. So not only does this *teacher-student* distillation technique maintain the accuracy of the baseline model, but it achieves real compression and with it practical increases in parsing speed and with a greener implementation. In absolute terms, our distilled models are faster than the previously fastest parser using sequence labelling, as can be seen explicitly in Table 4.1 for PTB, and outperforms it by over 1 point with respect to UAS and LAS when compressing to 40%. Distilling to 20% results in a speed 4x that of the sequence labelling model on CPU but comes at a cost of 0.62 points for UAS and 0.76 for LAS compared to the sequence labelling accuracies.

Furthermore, the increase in parsing accuracy for the smaller treebanks suggests that distillation could be used as a more efficient way of finding optimal hyperparameters depending on the available data, rather than training numerous models with varying hyperparameter settings.

			Full	D-20	D-40	D-60	D-80
gr	CPU	(tok/s)	1211 ± 2	2842 ± 3	2086 ± 3	1638 ± 3	1390 ± 1
		(sent/s)	75.4 ± 0.1	177.1 ± 0.2	130.0 ± 0.2	102.1 ± 0.2	86.6 ± 0.0
	GPU	(tok/s)	19219 ± 77	21017 ± 142	21296 ± 122	20346 ± 70	19202 ± 147
		(sent/s)	1197.6 ± 4.8	1309.6 ± 8.9	1327.0 ± 7.6	1267.8 ± 4.3	1196.5 ± 9.1
zh	CPU	(tok/s)	1124 ± 2	2503 ± 3	1872 ± 2	1490 ± 2	1278 ± 1
		(sent/s)	46.8 ± 0.1	104.2 ± 0.1	77.9 ± 0.1	62.0 ± 0.1	53.2 ± 0.0
	GPU	(tok/s)	21255 ± 113	25665 ± 82	24862 ± 134	23567 ± 28	22663 ± 91
		(sent/s)	884.7 ± 4.7	1068.3 ± 3.4	1034.9 ± 5.6	981.0 ± 1.2	943.4 ± 3.8
en	CPU	(tok/s)	884 ± 1	2217 ± 10	1548 ± 3	1217 ± 1	1010 ± 7
		(sent/s)	73.2 ± 0.1	183.5 ± 0.8	128.1 ± 0.3	100.7 ± 0.1	83.6 ± 0.6
	GPU	(tok/s)	16942 ± 25	20538 ± 60	19739 ± 109	19003 ± 90	17511 ± 57
		(sent/s)	1402.2 ± 2.1	1699.8 ± 5.0	1633.7 ± 9.0	1572.7 ± 7.4	1449.2 ± 4.7
fi	CPU	(tok/s)	988 ± 1	2586 ± 3	1767 ± 2	1371 ± 2	1153 ± 0
		(sent/s)	72.9 ± 0.0	190.9 ± 0.2	130.4 ± 0.2	101.2 ± 0.1	85.1 ± 0.0
	GPU	(tok/s)	18325 ± 46	22181 ± 50	21408 ± 130	20220 ± 90	19013 ± 33
		(sent/s)	1352.4 ± 3.4	1637.0 ± 3.7	1580.0 ± 9.6	1492.3 ± 6.7	1403.2 ± 2.4
he	CPU	(tok/s)	1180 ± 1	2644 ± 3	1964 ± 2	1582 ± 1	1337 ± 1
		(sent/s)	47.2 ± 0.0	105.7 ± 0.1	78.5 ± 0.1	63.3 ± 0.0	53.5 ± 0.0
	GPU	(tok/s)	22202 ± 98	26441 ± 150	25418 ± 181	24233 ± 176	22651 ± 89
		(sent/s)	887.4 ± 3.9	1056.8 ± 6.0	1016.0 ± 7.2	968.6 ± 7.1	905.4 ± 3.5
ru	CPU	(tok/s)	734 ± 1	1717 ± 3	1237 ± 1	976 ± 1	832 ± 1
		(sent/s)	38.7 ± 0.0	90.6 ± 0.1	65.3 ± 0.1	51.5 ± 0.1	43.9 ± 0.1
	GPU	(tok/s)	16383 ± 87	19661 ± 137	18337 ± 44	17901 ± 65	17014 ± 21
		(sent/s)	864.9 ± 4.6	1037.9 ± 7.2	968.0 ± 2.3	944.9 ± 3.4	898.2 ± 1.1
ta	CPU	(tok/s)	1110 ± 2	2334 ± 5	1799 ± 1	1464 ± 2	1251 ± 2
		(sent/s)	67.0 ± 0.1	140.8 ± 0.3	108.5 ± 0.1	88.3 ± 0.1	75.5 ± 0.1
	GPU	(tok/s)	17188 ± 194	19829 ± 126	19771 ± 106	18540 ± 98	18172 ± 151
		(sent/s)	1037.0 ± 11.7	1196.3 ± 7.6	1192.8 ± 6.4	1118.6 ± 5.9	1096.4 ± 9.1
ug	CPU	(tok/s)	1058 ± 1	2289 ± 3	1806 ± 2	1404 ± 2	1199 ± 2
		(sent/s)	92.2 ± 0.1	199.4 ± 0.3	157.3 ± 0.2	122.4 ± 0.2	104.5 ± 0.1
	GPU	(tok/s)	17974 ± 35	21298 ± 82	21004 ± 93	19738 ± 70	18963 ± 132
		(sent/s)	1566.0 ± 3.0	1855.6 ± 7.2	1829.9 ± 8.1	1719.6 ± 6.1	1652.1 ± 11.5
wo	CPU	(tok/s)	1245 ± 2	2559 ± 5	2021 ± 3	1614 ± 2	1398 ± 2
		(sent/s)	56.3 ± 0.1	115.6 ± 0.2	91.3 ± 0.1	72.9 ± 0.1	63.2 ± 0.1
	GPU	(tok/s)	20225 ± 74	24361 ± 94	21564 ± 73	20661 ± 102	21059 ± 105
		(sent/s)	913.8 ± 3.4	1100.6 ± 4.2	974.2 ± 3.3	933.4 ± 4.6	951.4 ± 4.7
avg	CPU	(tok/s)	1070 ± 21	2440 ± 39	1808 ± 32	1431 ± 26	1218 ± 23
		(sent/s)	63.5 ± 2.1	146.8 ± 5.4	108.1 ± 3.8	85.3 ± 2.9	72.5 ± 2.4
	GPU	(tok/s)	18933 ± 243	22503 ± 307	21488 ± 271	20463 ± 251	19666 ± 252
		(sent/s)	1124.7 ± 33.3	1336.3 ± 40.1	1282.8 ± 41.7	1220.4 ± 38.8	1168.2 ± 34.8

Table 4.8: Speeds with batch size 256.



(a) (b)

Figure 4.5: Comparison of attachment scores and percentage increase of speed (tok/s) for different distilled models with batch size 256: speed on GPU (a) and speed on CPU (b). Shaded areas show the standard error. Speeds for Tamil-TTB are not included as the test treebank is too small for larger batch sizes.

We also need to consider training costs, an important factor to implement green AI. In this respect, while our full baseline model took 66.4 seconds per epoch to train on English-EWT (the largest treebank used in this analysis), the baseline reduced to 20% trainable parameters required 52.9s per epoch, and the distillation into 20% of the original parameters clocked in at 103.1s per epoch. The distillation process takes longer and must be done after a full model is trained. However, the optimal model when distilling often occurred earlier (about epoch 50, rather than 80-100) suggesting less training is required.

In practice, the intended use of a parser should be considered when evaluating the environmental adequacy of distillation: in systems that will parse at a large scale or be deployed for extended periods of time, the savings at decoding time will offset the increased carbon footprint from training, but this may not be true in smaller-scale scenarios. However, in the latter, distillation can still be useful to reduce hardware requirements of the machine(s) used for decoding, indirectly reducing emissions.

4.4.3 Summary

We have obtained results that suggest using *teacher-student* distillation for UD parsing is an effective means of increasing parsing efficiency. The baseline parser used for our experiments was not only accurate but already fast, meaning it was a strong baseline from which to see improvements. We obtained parsing speeds 2.30x (1.19x) faster on CPU (GPU) while only losing ∼1 point for both UAS and LAS when compared to the original sized model. Furthermore, the smallest model which obtains these results only has 20% of the original model's trainable parameters, vastly reducing its environmental impact.

4.5 Experiment 2: EUD Parsing Shared Task

Here we describe our contribution to the Enhanced Universal Dependencies (EUD) Shared Task at IWPT 2020 (Bouma et al., 2020). We engaged with the task by focusing on efficiency while attempting to maintain accuracy. For this we considered both the training costs and the inference efficiency. We present this work here because the main model component of our system used distillation, so this offers a second data point on the usefulness of distillation for dependency parsing. We also describe the nature of the enhanced graph structure and the rule-based system we utilised in order to take part in the task.

Our models were a combination of distilled neural dependency parsers and a rule-based system that projects UD trees into EUD graphs, therefore combining linguistics and machine learning to develop efficient parsers. We also limited the amount of training data used. Despite focusing on efficiency, we obtained an average ELAS (the labelled attachment score of the EUD graph using enhanced relation labels) of 74.04 for our official submission, ranking 4th overall.

4.5.1 Details of dependency enhancements

In this section we give more details about the enhancement of dependency relations and about the processing subtleties of relative clauses, controlled predicates, and conjunctions.

Most of the original dependencies are kept in the enhanced structure, but they can undergo a number of cosmetic changes. In the simplest case, the relation type t is just appended to the index h of the word's governor to give the relation $h : t$. Sometimes, during the process the relation type is slightly modified. In Estonian (EDT and EWT) some complex relations such as `compound:prt` or `csubj:cop` are truncated and only the first part is kept. Conversely, in French (Sequoia) some relations receive extra information, such as subjects of passives `nsubj:pass` that are augmented with `xoxobj` stating they are the semantic object of their head.

Some relations receive extra lexical and morphological information. Conjuncts marked with `conj` usually receive the lemma of the coordinating conjunction (`cc`). Likewise, adverbial and

adjectival clauses (`advcl` and `acl`) receive the lemma of the word (`mark`) that introduces them. Nominal modifiers and obliques (`nmod` and `obl`) can receive the lemma of the adposition that introduces them (often marked with the `case` relation). Furthermore `nmod` and `obl` can also receive case information about the word itself. When the introducing marker is not a word but a fixed expression such as *"as well as"* then the *long lemma* composed of the lemmas of each word in the expression (marked by the `fixed` relation) is used, for example `conj:as_well_as`.

Relative clauses The only relations from the original tree that are not kept in the enhanced structure are those whose dependent is an anaphoric pronoun or adverb used to introduce a relative clause. Instead, the dependent (pronoun or adverb) is linked to its antecedent by an edge labelled `ref`. A new edge is then added between the original head of the reference and its antecedent of the same type as the original relation in order to show the argument structure of the clause. Thus, relative clauses are the first phenomenon that creates edges that are not present in the original tree. Their structure is however relatively simple since they can at most create one extra edge and replace one.

There are nonetheless two subtleties with relative clauses. First, in some languages, such as English, relative pronouns are not necessary. In these cases, while there are restrictions on the role the antecedent can fill, we need to infer its actual role from the sentence. Second, there may be several words that look like relativisers in a relative clause even outside conjunction. Often, only one of them is a leaf node, the others introducing further embedded clauses. Only in Finnish (TDT) did we find instances of multiple relative pronouns attaching to the same verb and each being marked as the reference of another word in the sentence.

Control A second phenomenon that creates new dependencies is control, where the subject of an embedded clause is not overt and is provided by one of its governor's arguments. For example in the English sentence *"I want you to go,"* the semantic subject of the verb *go* is the object of the main verb, namely *you*. In such a case, an additional relation is added to the structure to represent the dependency of the word *you* to the embedded predicate *go*. These structures are marked by a `xcomp` relation between the embedded predicate and its governor in the original tree. The identity of the new subject depends usually on the governing predicate and its argument structure. So it is mostly a matter of knowing the governing profile of each lexical item given their argument structure. For example, the subject of a predicate embedded in a *want to* clause is the object of the *want to* clause if present, its subject otherwise. Control is also quite simple since it has a limited span.

Conjunction The vast majority of new edges are created by conjunctions and this is much harder to handle than the two previous phenomena. Contrary to relative clauses and control, conjunction has no direction in the sense that it can occur both at the governor level and at the dependent level. In *"Mary and Sam bought strawberries,"* the conjunction *"Mary and Sam"* occurs at the dependent level and both *Mary* and *Sam* are subject of the verb *bought*. In *"Mary bought strawberries and ate them,"* the conjunction is now at the governor level and *Mary* is the subject of both *bought* and *ate*. So unlike relative clauses where one merely needs to find the relativiser's antecedent higher up in the tree, or control where one needs to look for the controlled subject amongst the arguments of the controlling predicate, conjunctions can have repercussions both higher up and lower down in the structure at the same time.

The easiest case for conjunction is when it occurs at the dependent level. One just needs to propagate the relation existing between the head of the conjunction and its governor to the other conjuncts. In the case of conjunction at the governor level, things are more complicated. While dependents don't tend to propagate up a conjunction chain but only down, they can be blocked by a number of reasons. For example in *"Mary bought and ate strawberries,"* the object *strawberries* should attach to *bought* in the tree and only propagate down to *ate*. But in *"Mary spoke and ate strawberries," strawberries* should attach to *ate*

and not propagate up to *spoke*, even though *speak* can also have direct objects. And in *"Mary bought strawberries and ate," strawberries* does not propagate down to *ate* since it appears before it in the sentence. However, the conditions under which certain dependents do or do not propagate to their governor's conjuncts are both language and relation specific. In a given language, objects need not behave like subjects nor like determiners or adverbials. Often if a relation slot (object, subject, determiner) is already filled for a given word, it will block the propagation of the same relation from higher up in the conjunction chain, but it need not always be the case, especially with adverbials. But even an empty slot does not always guarantee propagation, especially in case marking and prodrop languages where morphological considerations play a major role as well. So we need to learn the propagation conditions for each relation type on a per language basis.

In our system, we keep track of dependents of `conj` relations during the first traversal of a sentence and handle them in the second pass. The main reason for not processing conjuncts as soon as they arrive in the sentence is that some of their dependents (objects, adjectives or adverbials) can appear later and thus would require extra processing. For example, in *"Mary bought and ate strawberries,"* the object of both verbs only appears after the conjunct *ate*, so upon first seeing *ate*, *bought* does not have any object to be propagated.

4.5.2 Limiting training data

Distillation introduces extra training overheads. To mitigate this and to balance our pursuit of inference efficiency with some semblance of training efficiency and considering the results in Section 4.4 suggest that distillation for larger treebanks causes a larger drop in accuracy, we decided to set a limit to the size of training treebanks.

In order to minimise introducing compounding variables that could affect training efficacy, we renormalise the sampled treebanks to follow the same sentence length distribution of the original treebank. Where more than one treebank exists for a given language, we took a sample from each treebank renormalised with respect to that treebank and took a sample size so that the contribution from each treebank would follow the same ratio as the full data for that language.

We evaluated what limit to set by testing on 4 languages spanning 3 language families (Uralic, Afro-Asiatic, and Indo-European). The only family to appear in the shared task training data not covered was Dravidian as the only example from this language, Tamil, has too small a treebank to have been useful for this analysis. We also cover two branches of the Indo-European family. Balto-Slavic is covered by Russian as the treebank is rather large and uses the Cyrillic script. Germanic is covered by Dutch, which we chose as there are two treebanks which combine to a sizeable number of trees and so would cover the case of combining different treebanks. Finnish was used to cover the Uralic family as we carried this experiment out before the larger Estonian treebank was made available and Arabic was used for Afro-Asiatic. We used sample treebank sizes of 1,000, 3,000, 6,075 (the number of trees in the Arabic treebank), 12,217 (the number of trees in the Finnish treebank), and 18,051 (the combined number of trees in both Dutch treebanks). We created 2 splits where possible (i.e. at 6,075 trees Arabic isn't a sample treebank) as a limited attempt at experimental robustness.

We train a biaffine parser using the hyperparameters of the original paper, shown in Table 4.9. We then distill (as described in Section 4.3) these models to two different network sizes, one which has 70% of the number of nodes in both the BILSTM and MLP layers and one that has 50%. Otherwise the structure of the network is the same as the base model. The LAS averaged over the splits for each sample and model are shown in Figure 4.6 and for UAS in Figure 4.7. We are limited by what we can extrapolate from the results for Arabic and Finnish other than they appear to follow a similar trend to Dutch and Russian. For the latter languages we observe the performance levelling at larger treebank sizes, which is neither remarkable nor unexpected, but also a widening between the performance of the full

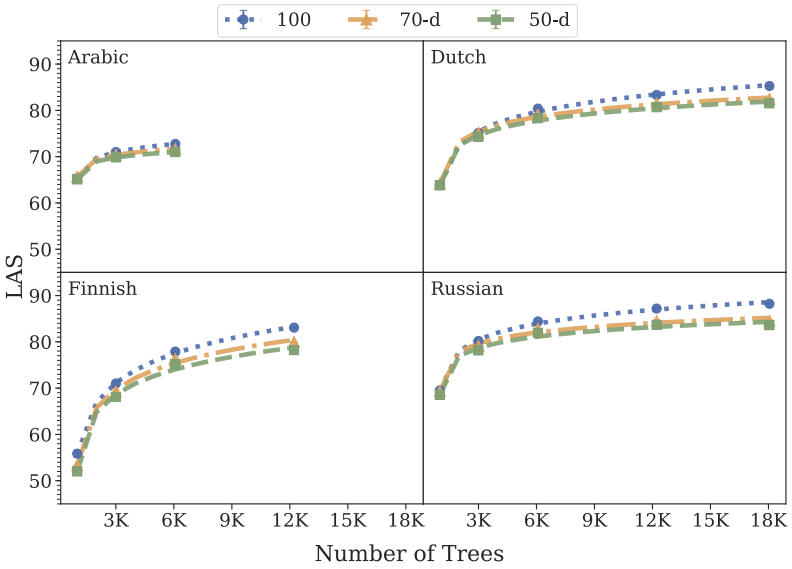


Figure 4.6: LAS for different models for Arabic, Dutch, Finnish, and Russian development treebanks.

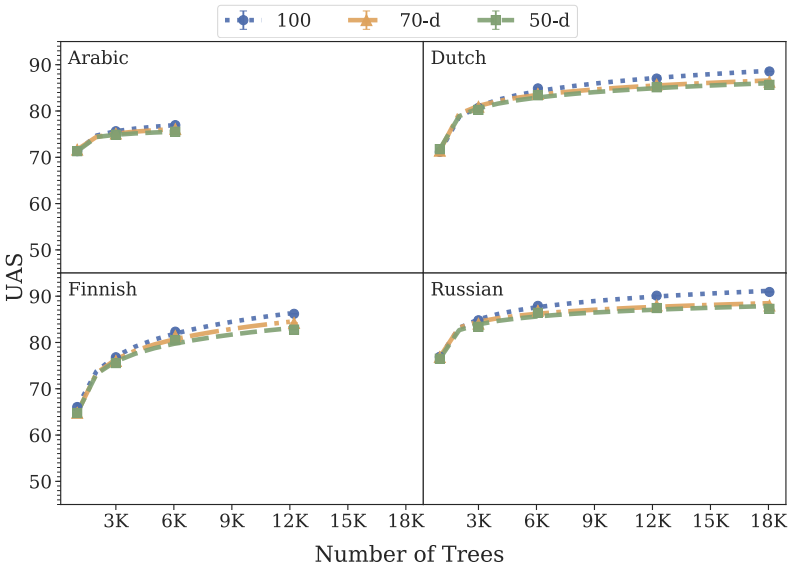


Figure 4.7: UAS for different models for Arabic, Dutch, Finnish, and Russian development treebanks.

and the distilled models.

hyperparameter	value
word embedding dimensions	100
char embedding dimensions	32
char BiLSTM dimensions	100
embedding dropout	0.33
BiLSTM dimensions	400 (200)
BiLSTM layers	3
arc MLP dimensions	500 (250)
label MLP dimensions	100 (50)
MLP layers	1
learning rate	0.2
dropout	0.33
momentum	0.9
L2 norm λ	0.9
annealing	$0.75^{\wedge}(t/5000)$
ϵ	1×10^{-12}
optimiser	Adam
loss function	cross entropy
epochs	100
min vocab freq.	2

Table 4.9: Hyperparameters for baseline models. The values in parentheses show the values for the distilled and small models used in the main analysis of the shared task.

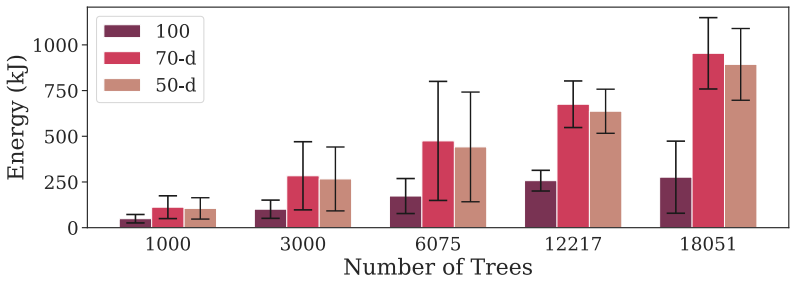


Figure 4.8: Training energy consumption for different models for different treebank sizes averaged over Arabic, Dutch, Finnish, and Russian.

As we are concerned with training efficiency, we present the energy consumption for each model type averaged over language and split in Figure 4.8. The amount of energy required to distill our models increases significantly with respect to treebank size. However, distilling to a smaller model requires less energy and, as can be seen in Figure 4.6, the accuracy difference between the two distilled models is nominal.

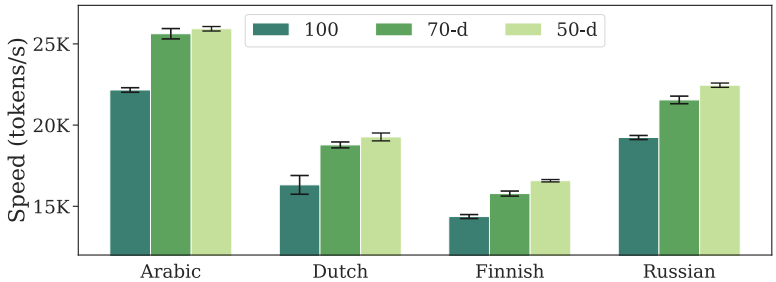


Figure 4.9: GPU inference speed for different models for treebank 12k (except Arabic which uses its full treebank of 6075 trees) averaged over 5 runs on the development treebanks with batch size 256.

Figure 4.9 shows the inference speed (averaged over splits and 5 runs) on GPU using a single CPU core for each language using the models trained with the 12,217-sentence treebanks (for Arabic we use its full treebank). We observe a sizeable increase in speed over

the baseline model for both distilled models, but only a small difference between the two distilled models.

From this, we decided to set an upper limit on the treebank size for the main task to 13,121 (the size of the Italian treebank) as this would require the least amount of tampering and was close to the second largest treebank size used here which performed close to the largest. This meant taking a sample of the Czech, Dutch, Estonian, and Polish treebanks and combining them as described above. A sample was taken for the Russian treebank. Some syntactic metrics are given in Table 4.10 which shows the different breakdown of the training data used for each of these languages and how they are very similar to the full data. Also, we opted to distill to 50% of the original model size. For this analysis, and all subsequent analyses, the CPU used was an Intel Core i7-7700 and the GPU an Nvidia GeForce GTX 1080.[5]

	original				sample				
	trees	mL	mDD	NP%	trees	mL	mDD	NP%	2s-KS$_L$
Czech									
-CAC	23478	20.1	3.7	2.5	3016	19.9	3.7	2.5	0.014
-FicTree	10160	13.2	3.6	3.8	1305	13.1	3.6	3.8	0.016
-PDT	68495	17.1	3.7	2.7	8800	17.1	3.7	2.6	0.007
-combined	102133	17.4	3.7	2.7	13121	17.4	3.7	2.7	0.007
Dutch									
-Alpino	12264	15.2	4.0	4.5	8915	15.2	4.0	4.4	0.004
-LassySmall	5787	13.0	3.7	2.0	4206	13.0	3.7	1.9	0.007
-combined	18051	14.5	3.9	3.8	13121	14.5	3.9	3.7	0.004
Estonian									
-EDT	24633	14.0	3.6	0.8	12552	14.1	3.6	0.8	0.010
-EWT	1116	15.4	3.8	1.5	569	15.4	3.8	1.6	0.027
-combined	25749	14.1	3.6	0.9	13121	14.1	3.6	0.8	0.009
Polish									
-LFG	13774	7.6	2.8	0.3	5738	7.6	2.8	0.3	0.006
-PDB	17722	15.9	3.4	1.4	7383	15.9	3.4	1.5	0.008
-combined	31496	12.3	3.3	1.1	13121	12.3	3.3	1.2	0.003
Russian									
-SynTagRus	48814	17.8	3.6	1.6	13121	17.8	3.6	1.6	0.004

Table 4.10: Analysis of renormalised treebank samples: 2s-KS is the two-sample Kolmogorov-Smirnov test comparing the sentence-length distributions of the original and the sample treebanks (where values close to 0 suggest samples are not from different distributions, and values approaching 1 suggest otherwise); trees is the number of trees; mL is the mean sentence length; mDD the mean dependency distance; and NP% is the percentage of non-projective arcs. Where we use the combined sample (or just the sample for Russian-SynTagRus) for training.

4.5.3 Distilled dependency parsers

We extend the work of Section 4.3 and use *teacher-student* distillation to obtain efficient dependency parsers as the basis of our enhanched-dependency parser systems.

While we curtailed our training data, we selected our models based on the performance on the full development data for a given language with gold sentence segmentation and tokenisation. We used characters and words as input to our network. The embeddings for both were randomly initialised. The hyperparameters are the same as used above (Table 4.9). We also used early stopping to limit unnecessary training time, stopping after 10 epochs without performance improvement.

At inference time we used UDPipe v2.5 models to predict everything except the parse (Straka and Straková, 2019b). When a combination of treebanks were being predicted, we used the model which corresponded to the largest of the treebanks.

[5]Using Python 3.7.0, PyTorch 1.0.0, and CUDA 8.0.

Training costs		
	Total time	GPU Energy (kJ)
Base	08h:42m:52.1s	3570.7
Distill	30h:07m:49.6s	9981.8
Rule-based	00h:00m:41.1s	*n/a*

Table 4.11: Total training time and GPU energy consumption for all treebanks.

Table 4.11 shows the total time to train the full-sized models and the distillation models for all languages. Also, shown is the GPU energy consumption. The costs for distillation include those of the base models.

Training costs for distillation are more than three times that of the baseline which is hardly surprising. The inference energy cost for all development treebanks (37K trees) for the full model is 2.10 (0.09)kJ (average value over 5 runs for each treebank) whereas the cost for distillation is 1.49 (0.03)kJ. Based on these measurements, we would need to parse 390M sentences to offset the extra cost of distilling models when running on GPU.

	UAS	LAS		UAS	LAS
Arabic			***Bulgarian***		
small	76.9	72.5	small	91.6	87.6
dist	76.5	72.3	dist	91.6	87.6
Czech			***Dutch***		
small	89.5	86.0	small	87.2	83.3
dist	89.0	85.3	dist	86.7	82.9
English			***Estonian***		
small	85.0	81.9	small	85.2	80.9
dist	84.4	81.2	dist	84.7	80.2
Finnish			***French***		
small	85.8	82.2	small	88.1	85.5
dist	85.1	81.3	dist	88.5	85.8
Italian			***Latvian***		
small	91.3	89.0	small	86.3	82.4
dist	90.3	87.8	dist	86.0	81.9
Lithuanian			***Polish***		
small	76.7	71.5	small	90.5	86.4
dist	78.0	73.0	dist	90.2	86.0
Russian			***Slovak***		
small	89.5	86.3	small	85.6	81.7
dist	88.9	85.5	dist	84.7	80.7
Swedish			***Tamil***		
small	84.5	80.8	small	63.7	55.7
dist	85.3	81.6	dist	64.0	56.9
Ukrainian			***Average***		
small	86.8	82.6	small	85.0	80.9
dist	86.6	82.5	dist	84.7	80.7

Table 4.12: Comparison of attachment scores for the development treebanks for distilled (dist) models and models with the same parameters (small) trained normally.

Late in the day we decided to validate the results of Section 4.4, namely that distilled models outperform models trained normally of equivalent sizes. This highlighted that our distilled models used for our official score had not converged. We trained new distilled models and the results given here are for these new models. Our official results using the partially-trained models are in table 4.17 at the end of this chapter. All results, including training costs, in this section are for the full-trained distilled models and unless otherwise stated are using the combined development treebanks for each language.

Table 4.12 shows the performance for the equivalent-sized models trained normally (small) and the distilled models (dist) with respect to UAS and LAS. For the most part the normal models outperform the distilled models. The main differences between this experimental

setting and that in Section 4.4 is we do not use pre-trained word embeddings nor POS tags as features. So perhaps without this extra information distillation is less effective. Also, dropout wasn't used during distillation in said section but is here, so perhaps the values used here were too punitive a regularisation. Although we use the same hyperparameters as the original experiment, the average LAS for the small normally trained models is 0.4 points less than the large model.

	UAS	LAS	ELAS		UAS	LAS	ELAS
Arabic				***Bulgarian***			
full	77.0	72.8	68.4	full	91.5	87.6	85.3
dist	76.5	72.3	67.9	dist	91.6	87.6	85.2
udpipe	72.8	68.1	63.0	udpipe	88.7	84.3	81.9
Czech				***Dutch***			
full	90.0	87.0	82.4	full	87.5	84.0	82.2
dist	89.0	85.3	80.7	dist	86.7	82.9	81.0
udpipe	87.6	84.0	78.4	udpipe	79.3	75.0	73.2
English				***Estonian***			
full	85.6	82.6	81.2	full	85.5	81.5	80.3
dist	84.4	81.2	79.8	dist	84.7	80.2	79.0
udpipe	81.0	77.6	76.3	udpipe	81.5	77.6	76.7
Finnish				***French***			
full	86.2	83.1	79.9	full	88.1	85.5	82.3
dist	85.1	81.3	78.0	dist	88.5	85.8	82.6
udpipe	80.4	76.8	73.7	udpipe	85.2	82.6	79.4
Italian				***Latvian***			
full	91.6	89.3	87.8	full	86.7	83.2	79.3
dist	90.3	87.8	85.9	dist	86.0	81.9	78.2
udpipe	88.5	85.9	84.1	udpipe	79.8	75.4	70.5
Lithuanian				***Polish***			
full	77.6	72.7	68.6	full	90.9	87.2	78.6
dist	78.0	73.0	68.9	dist	90.2	86.0	77.2
udpipe	72.3	64.6	60.9	udpipe	87.1	82.6	74.7
Russian				***Slovak***			
full	90.2	87.3	84.4	full	85.4	81.6	77.0
dist	88.9	85.5	82.5	dist	84.7	80.7	76.1
udpipe	87.4	84.4	81.5	udpipe	81.2	75.9	70.5
Swedish				***Tamil***			
full	85.2	81.4	78.9	full	59.8	52.6	51.2
dist	85.3	81.6	79.0	dist	64.0	56.9	55.5
udpipe	79.5	75.4	73.2	udpipe	60.7	54.1	53.0
Ukrainian				***Average***			
full	87.1	83.2	78.3	full	85.0	81.3	78.0
dist	86.6	82.5	77.5	dist	84.7	80.7	77.3
udpipe	81.6	76.9	72.5	udpipe	80.9	76.5	73.1

Table 4.13: Attachment scores for both UD trees and EUD graphs for the development treebanks using different dependency parsers: full baseline models (Full), distilled models (dist), and UDPipe v2.5 models (udpipe).

We also evaluated the distilled models against the full baseline model and UDPipe v2.5. These results are shown in Table 4.13. The distilled models outperform the UDPipe models and are within a point of both UAS and LAS to the full model. The ELAS results for the rule-based system using the predicted dependency trees from each of these systems are also shown. The performance on ELAS generally follows the dependency scores.

Figure 4.10 shows the inference speed using GPU and CPU of the full baseline model and the distilled models for each language. These are obtained by running the parser 5 times for each language on the full development data and only using one CPU core. The average speed (token/second) increase was 2.44x (1.17x) on CPU (GPU).

Table 4.14 shows the inference speeds for the full pipeline and the dependency parser. We also compare UDPipe inference performance as it is a viable candidate for an efficient parser. It is the fastest of the systems compared here, but the full pipeline which used it obtained

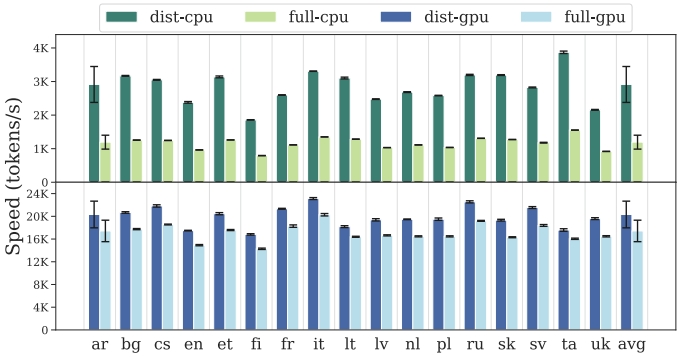


Figure 4.10: Inference speed for distilled (dist) and full baseline models on CPU (-cpu) and GPU (-gpu) for each development treebank averaged over 5 runs using one CPU core with batch size 256.

		Inference speed (token/s)	
		UD parser	Full pipeline
CPU	**Base**	1194.1 (207.1)	879.0 (123.4)
	Distill	2912.9 (535.1)	1569.9 (238.8)
	UDPipe	3629.4 (584.0)	2220.2 (698.0)
GPU	**Base**	17427.0 (1890.3)	2993.3 (680.2)
	Distill	20321.6 (2348.9)	3073.7 (714.9)

Table 4.14: Inference speeds for dependency parsers and the full EUD pipeline for different systems run on development treebanks and averaged over 5 runs.

an average ELAS 4.9 points less than the full baseline model whereas the distilled models are only 0.7 points less.

4.5.4 Rule-based conversion

Rule-based systems are intrinsically efficient with respect to training time (barely a flash in the pan) and inference time (there is practically none). So we developed a simple rule-based system to enhance the existing dependency tree and reveal hidden dependencies in a cross-lingual setting using as few language specific rules as possible. Beyond the basic enhancement of the original dependencies, there are four main phenomena that create new dependencies: relative clauses, control, conjunction and ellipsis. Since our pipeline does not predict empty nodes, we decided to ignore ellipsis in this system. To deal with each of these phenomena, our algorithm needs to make a number of passes over each sentence.

Pass one - relative clauses and controls: The first pass of the algorithm iterates through each word in the sentence and creates enhanced relations according to the type of the original dependency. When necessary, it adds lemma and case information. If the current word is a relative pronoun/adverb, its antecedent is found by following its path to the root until an `acl:relcl` relation is met. Then a `ref` edge is created between the word and its antecedent and an edge between the antecedent and the governor of the relativiser with the same relation type as the original relation (if the relative pronoun is the object of a verb then the antecedent becomes the object of that verb). If the word is the dependent of an `xcomp` relation, the algorithm looks for a subject amongst its controlling predicate's arguments. If a subject is found, it creates an edge between the subject and the current word of type `nsubj(:xsubj)` (or `csubj` in the case of a clausal argument). If no subject is available, the current word is stored in a separate list for later processing. If the word is the dependent of a `conj` relation, it too is stored in a separate list along with all other conjuncts. Whenever we encounter an argument of the type subject, object or oblique, this information is kept for resolving subjects of controlled predicates.

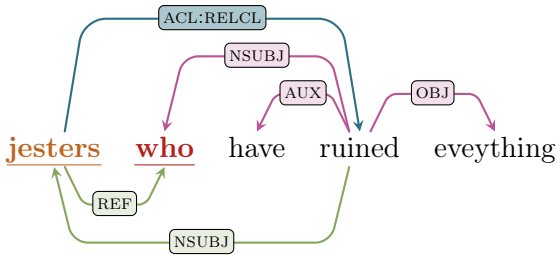


Figure 4.11: Relative clause example. Pre-existing edges in graph are in magenta and blue. The algorithm observes an `acl:relcl` relation (highlighted in blue) which causes it to generate two new relations (highlighted in green). A `ref` relation is created between **who** and its antecedent, **jesters**. Then a `nsubj` is propagated from the head of **who**, *ruined*, to **jesters**.

Pass two - resolving conjunctions: We have two general functions, one for dependent level conjunctions and one for governor level conjunctions, and a few special cases. The dependent level function propagates the conjunction head's original relation to its conjuncts adapting it if necessary, for example in coordinated `nmod` with different adposition or case. The governor level function propagates the conjunction head's dependents to its conjuncts in the absence of similar dependents and according to morphological agreement. We have a special function that handles subjects of conjuncts because subjects are more diverse than other syntactic functions. In UD at least three relations can mark subjects, namely `nsubj` for nominal subjects, `csubj` for clausal subjects and `expl` used amongst other for syntactic subjects in non prodrop languages (e.g. *"it rains"*). Subject edges also embed information about their governor, notably information about the voice as `:pass` when relevant. And, subjects can be absent altogether in prodrop languages, so we rely on morphological information to decide to propagate a given subject in these languages.

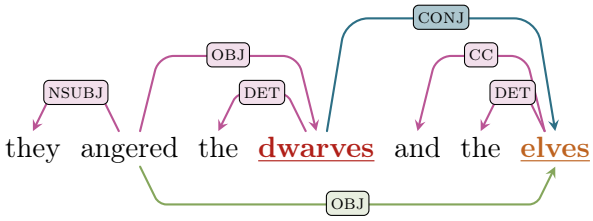


Figure 4.12: Conjunction example. Magenta and blue edges are those existing in the graph after one pass. During the first pass **elves** is stored as it is the dependent of a `conj` relation (highlighted in blue). On the second pass the `obj` relation of **dwarves**, the head of this `conj` relation, propagates to **elves** generating a new `obj` relation (highlighted in green) from *angered*.

Pass three and onwards - sweeping up controls: Once conjunctions have been resolved and more predicates have their arguments stored, the algorithm iterates over controlled predicates that do not have a subject after the first sentence traversal. Several such iterations may be necessary since the number of times a predicate may be coordinated with a controlled verb itself already coordinated to another controlled verb is not bounded. Like in the sentence *"Sam stood up and wanted to scream and start running."* But in practice one iteration solves the vast majority of missing subjects.

4.5.5 Tuning the rules

A number of enhancements are relation and language specific and some even lexically conditioned such as control, and not all languages include every enhancement type. So the training data is used to tune rules to a given language while keeping the rule definitions as generic as possible.

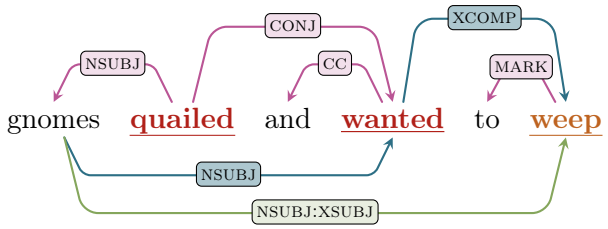


Figure 4.13: Control example. The edges of the graph after two passes are in magenta and blue. During the first pass **weep** is stored as it is a dependent of a `xcomp` relation (highlighted in blue) but it cannot be resolved until **wanted** is. **wanted** is resolved in the second pass and an `nsubj` relation (shown in blue) is propagated from the head, *gnomes*, of its conjunct, **quailed**. In the third pass this is further propagated to **weep** generating a `nsubj:xsubj` relation (highlighted in green).

The first type of information needed regards additional lemmas and cases appearing in edges. For each relation type, the frequency at which `case` is being added to the relation is obtained. Similarly for lemma, the algorithm counts the frequency of relation types between a word and its dependent used for lexicalisation since different relations are augmented with different dependents (`obl` usually uses `case` where `acl` prefers `mark`). Furthermore, for lemmas, when several dependents have the same relation, it checks which is used for lexicalisation. For `conj` though, it only checks if there is anything at all since `conj` is tightly linked to `cc`.

Each language is tested to see if it is prodrop by comparing the number of root verbs with an overt subject to the number of root verbs without an overt subject. Whether `:xsubj` and `:relsubj` should be added to subjects of controlled predicates and relative clauses is also checked.

The algorithm then checks whether each relation propagates to its governor's conjuncts and under which conditions (the conditions are detailed in below) and also if it propagates to its own conjuncts. This is mostly relevant since `root` usually does not propagate to conjuncts of the main predicate, but in some treebanks it does.

Morphological features are used for detecting relativisers. For each morphological feature, the number of times it co-occurs with a `ref` enhanced relation is compared to the number of times it co-occurs with another relation. While not an arbitrary choice, it is one of the few cases where an enhanced relation does not depend directly on information in the original tree but on information external to the tree, so in theory we could have chosen other clues such as the lemma of the word instead. These pronouns and adverbs are usually marked with `PronType=Rel` or `PronType=Int,Rel`.

Finally, the controlling profile of controlling predicates is learnt. The system discerns which of the arguments is used as subject of controlled verbs and in which conditions, meaning that we do not count subjects in the absence of other arguments since they become default.

Conjunction propagating conditions We use two sets of conditions in order to guide the propagation of dependents to their governor's conjuncts. The first is about relation types already attached to these conjuncts. Usually an object or a subject does not attach to a verb that already has these slots filled. So for each relation, we measure three frequencies. The frequency at which it co-occurs with other types under its main governor (in the tree), the frequency at which it co-occurs with other types under its conjunct governors (in the enhanced structure) and the frequency at which it does not co-occur with other types because it does not propagate to its governor's conjunct. Any relation with which it co-occurs under its main governor cannot be blocking propagation. Then if a relation is more often than not associated with conjunct governors to which the current relation did not propagate, it is considered a blocking relation. In practice this means that a `subj` does not propagate to a conjunct of its governor that already has an `expl`, for example.

The second condition is based on matching morphological information. For every relation and morphological category (tense, case, aspect, and so on), we measure how often the value of a category agrees or disagrees between the governor and its conjunct (of the same UPOS tag) when the relation propagates and when it does not. If a category disagrees more often than not between conjuncts which the relation did not propagate, then we assume that the category needs to agree for that relation.

4.5.6 Problems

While our rule-based system performs remarkably well, as can be seen in Table 4.15, with the lowest ELAS being 94.9 on the gold development data, it is challenging to improve across languages simultaneously. Besides the expected ambiguity of language, there are several issues which limit us, some easy to fix, some more complicated, some language specific, and some more general.

	ELAS			
	Gold	Full	Dist	UDPipe
Arabic	98.8	68.4	67.9	63.0
Bulgarian	98.6	85.3	85.2	81.9
Czech	97.9	82.4	80.7	78.4
Dutch	98.9	82.2	81.0	73.2
English	99.5	81.2	79.8	76.3
Estonian	99.2	80.3	79.0	76.7
Finnish	97.3	79.9	78.0	73.7
French	98.9	82.3	82.6	79.4
Italian	99.5	87.8	85.9	84.1
Latvian	95.7	79.3	78.2	70.5
Lithuanian	98.8	68.6	68.9	60.9
Polish	94.9	78.6	77.2	74.7
Russian	98.6	84.4	82.5	81.5
Slovak	98.8	77.0	76.1	70.5
Swedish	98.8	78.9	79.0	73.2
Tamil	99.3	51.2	55.5	53.0
Ukrainian	95.8	78.3	77.5	72.5
Average	98.2	78.0	77.3	73.1

Table 4.15: Enhanced labelled attachment score for EUD graphs when using gold labelled dependency development treebanks (gold), predicted treebanks with full baseline models (Full), distilled models (Dist), and using UDPipe v2.5 models (UDPipe).

	Tokens	Words	Sentences	UPOS	XPOS	UFeats	AllTags	Lemmas	UAS	LAS	CLAS	MLAS	BLEX	EULAS	ELAS
Arabic	100.0	94.6	82.1	88.5	84.0	84.2	82.0	88.5	76.5	72.0	68.0	57.0	63.0	70.2	67.8
Bulgarian	99.9	99.9	94.2	97.6	94.3	95.4	93.8	94.6	92.1	88.5	84.5	78.0	77.5	87.3	86.4
Czech	99.9	99.9	93.2	97.8	90.9	90.8	89.7	97.4	88.0	84.1	80.9	70.4	78.6	82.0	79.6
Dutch	99.7	99.7	69.3	92.6	89.9	92.0	89.0	94.4	84.5	80.8	73.7	63.5	68.0	79.3	78.7
English	99.2	99.2	83.8	93.6	92.8	94.1	90.7	95.4	84.8	81.7	77.7	69.0	73.8	80.8	80.1
Estonian	99.7	99.7	90.0	95.0	96.2	92.8	91.0	90.4	82.7	78.2	75.5	67.3	66.2	77.7	76.8
Finnish	99.7	99.7	88.7	94.8	54.5	93.0	51.8	87.1	86.1	82.6	80.0	72.1	67.0	80.8	79.4
French	99.7	99.2	94.3	93.5	99.2	88.8	87.3	94.9	87.8	82.2	74.8	60.5	69.1	81.6	79.5
Italian	99.9	99.8	98.8	97.2	97.0	97.1	96.2	97.4	91.4	89.1	83.8	79.2	80.3	87.6	86.9
Latvian	99.3	99.3	98.7	93.5	84.3	89.5	83.9	92.7	86.0	81.8	78.7	65.9	72.4	79.3	77.8
Lithuanian	99.9	99.9	87.9	90.3	80.7	81.2	79.3	88.8	75.2	69.4	66.0	48.4	56.8	66.6	64.5
Polish	99.4	99.8	97.5	96.4	84.9	83.6	80.3	95.6	90.1	85.9	82.4	62.2	77.8	84.0	77.5
Russian	99.6	99.6	98.8	97.8	99.6	85.3	85.0	96.5	89.3	86.2	83.4	65.5	80.0	84.5	83.3
Slovak	100.0	100.0	85.3	92.9	77.1	80.3	76.7	86.6	85.6	81.5	78.0	56.8	64.8	79.8	76.7
Swedish	99.2	99.2	93.5	93.3	91.0	84.9	83.2	90.0	83.4	79.3	76.0	58.6	67.0	77.9	77.0
Tamil	99.2	94.5	97.5	81.3	76.3	80.5	75.6	84.1	62.5	53.0	48.8	39.9	43.7	53.0	51.7
Ukrainian	99.8	99.8	96.6	94.9	84.0	84.3	83.3	93.6	85.0	81.0	76.4	59.6	70.0	78.4	76.4
Average	99.7	99.1	91.2	93.6	86.9	88.1	83.5	92.2	84.2	79.8	75.8	63.2	69.2	78.3	76.5

Table 4.16: Test results evaluated through the official submission site and using our updated distilled model. Our official submission results can be seen in Table 4.17.

On the monolingual front, incomplete, erroneous and inconsistent annotations are the biggest problems. Incomplete annotation can occur both at the enhanced dependency and

	Tokens	Words	Sentences	UPOS	XPOS	UFeats	AllTags	Lemmas	UAS	LAS	CLAS	MLAS	BLEX	EULAS	ELAS
Arabic	100.0	94.6	82.1	88.5	84.0	84.2	82.0	88.5	75.8	71.2	66.8	56.1	62.0	69.2	66.9
Bulgarian	99.9	99.9	94.2	97.6	94.3	95.4	93.8	94.6	91.1	87.0	82.3	75.8	75.5	85.8	84.9
Czech	99.9	99.9	93.2	97.8	90.9	90.8	89.7	97.4	86.2	81.8	78.1	67.9	75.8	79.6	77.2
Dutch	99.7	99.7	69.3	92.6	89.9	92.0	89.0	94.4	83.4	79.4	71.9	61.9	66.4	78.0	77.4
English	99.2	99.2	83.8	93.6	92.8	94.1	90.7	95.4	83.7	80.1	75.8	67.2	72.1	79.2	78.5
Estonian	99.7	99.7	90.0	95.0	96.2	92.8	91.0	90.4	80.7	75.5	72.7	64.6	63.8	75.0	74.1
Finnish	99.7	99.7	88.7	94.8	54.5	93.0	51.8	87.1	84.1	79.7	76.5	69.0	64.3	77.8	75.7
French	99.7	99.2	94.3	93.5	99.2	88.8	87.3	94.9	87.2	80.6	72.1	58.3	66.7	80.1	77.8
Italian	99.9	99.8	98.8	97.2	97.0	97.1	96.2	97.4	90.2	87.4	81.4	76.8	77.9	85.9	84.8
Latvian	99.3	99.3	98.7	93.5	84.3	89.5	83.9	92.7	84.4	79.7	76.1	63.6	70.0	77.2	75.6
Lithuanian	99.9	99.9	87.9	90.3	80.7	81.2	79.3	88.8	72.9	66.3	62.6	45.9	54.3	63.7	61.4
Polish	99.4	99.8	97.5	96.4	84.9	83.6	80.3	95.6	88.4	83.4	79.4	60.1	75.0	81.4	74.5
Russian	99.6	99.6	98.8	97.8	99.6	85.3	85.0	96.5	86.8	83.2	80.0	62.8	76.7	81.7	80.3
Slovak	100.0	100.0	85.3	92.9	77.1	80.3	76.7	86.6	83.2	78.3	73.9	53.8	61.6	76.5	73.5
Swedish	99.2	99.2	93.5	93.3	91.0	84.9	83.2	90.0	82.2	77.6	73.7	56.8	64.9	76.2	75.2
Tamil	99.2	94.5	97.5	81.3	76.3	80.5	75.6	84.1	59.6	48.8	43.6	35.5	39.6	48.1	47.0
Ukrainian	99.8	99.8	96.6	94.9	84.0	84.3	83.3	93.6	83.4	78.7	73.6	57.8	67.4	76.2	74.0
Average	99.7	99.1	91.2	93.6	86.9	88.1	83.5	92.2	82.5	77.6	73.0	60.8	66.7	76.0	74.0

Table 4.17: Full test results for our official submission using the shared task's submission site for evaluation.

the lower level of annotation. For example in Dutch Alpino, we miss 515 `ref` relations and thus at least as many enhanced relations from their antecedents, representing two thirds of the missing dependencies. The bulk of these missed references are relative/interrogative pronouns/adverbs that are not annotated with an empty feature column. We wanted to avoid too many language specific rules and ignored them, leading to more than a thousand missing edges. Likewise, in some languages not all relativisers (typically interrogative adverbs) are marked as references when they should be according to UD guidelines.

Erroneous annotations can be at lower levels of annotation of the dependency tree, thus when applying rules according to these annotations, erroneous edges are created. For example in English (EWT), there is the sentence *"Let me know if this is the appropriate steps that you would like to see,"* in which *that* which references *steps* is analysed as the object of *like* (*"you would like the steps to see"* vs. *"you would like to see the steps"*) thus the controlling rule for *like* makes *steps* the subject of *see* in place of *you*. Annotation errors can also happen in the enhanced structure. In Russian, for example, a number of nominal modifiers have diverging case information in the feature column and in the enhanced relation one, often `Case=Gen` with `nmod:acc`, so the predicted enhanced relation `nmod:gen` conflicts with the actual annotation.

Latvian offers an example of inconsistent annotation, `nmod` is extended with either the adposition's lemma or the word's case but never both and the selection of lemma or case for any given word is seemingly arbitrary. So it is impossible to devise a rule to address this issue.

However, most of these problems are easily rectified with a system such as ours by checking the agreement of case and lemma information in enhanced relations assuming valid annotation of the underlying data. On the cross-lingual front, the biggest problem is lack of consistency in annotation conventions. Leaving incomplete annotation aside, there are a number of clear divergences. The most striking example is the way subjects of passive verbs and more generally enhanced relations are handled in French Sequoia. These relations receive an extra `(:)enh` to differentiate them from canonical relations directly taken from the tree, the presence of the column depends mostly on the number of columns in the relation type, if it is a simple relation then a column is used but when it is already a sub-type with a column between the main type and extra information then no column is added. Not only is this unique to this treebank, but it is also redundant since this information can be directly retrieved by looking at the original tree. There are also a number of more subtle inconsistencies. For example, in languages that add lemma information to `conj` relations, when the coordinating conjunction is a symbol (& or /), most languages just ignore them

and keep the bare `conj` relation. However, Swedish uses the special `conj:sym` relation.

Beyond these issues, there remain genuine linguistic difficulties. A difficulty common to all languages is the scope of conjunctions and whether to propagate dependents amongst conjuncts or not. This is particularly clear with adverbials and obliques that modify verbs. Due to their broad semantic range, adverbials can propagate from conjunction heads to dependent conjuncts even if they already have other adverbials, as long as they do not conflict semantically. Currently in UD, there is no hierarchy amongst dependents of a word, but there could be a form of scope indexing to distinguish a word's direct dependents from dependents of the whole conjunction attached to its head.

Another difficulty is subject selection in prodrop languages. Fortunately, the prodrop languages in this shared task have personal and number agreement at least on finite verbs which helps testing the compatibility of the overt subject of a verb with its coordinated verbs or verbs in relative clauses that lack an overt subject. However, there are prodrop languages that do not mark personal agreement on verbs and do not use relativisers either (e.g. Japanese). In this case, finding the semantic subject of verbs may be much more challenging.

Limits and issues

While being above 94.9 ELAS for all languages, our rule-based system could still be improved to better capture enhanced structures. There are three main points for further improvement.

Upon reviewing the code for the rule-based system, we realised that we catch arguments of relative clauses only in presence of a relativiser that receives the `ref` relation. This means that we miss a number of relations involved in relative clauses. It remained unnoticed because of all the languages in the shared task, most use relative pronouns/adverbs to introduce relative clauses. In fact the only language that does not have relative pronouns, Tamil, is not yet annotated with relative clauses and it might not even be relevant. Our methodology here is to look for an antecedent when we have a relative pronoun, but we could do the opposite and look for potential relative pronouns when we have a relative clause. The latter should indeed be more language agnostic and work even when there are no relativisers involved.

A second point of improvement has to do with subject finding in controlled predicates. In our current system, the controlling behaviour of each controlling construction is gathered from the training data, and if we encounter an out of vocabulary construction at prediction time the subject is used by default. But further consideration showed that the object might be a more sensible default option when available. It would, however, be more interesting to learn the default behaviour on a per language basis.

Thirdly, due to the march of time, we hard-coded a number of heuristic thresholds used to fine-tune the system. For example, to see if a language is prodrop, we compare the number of root verbs with overt subjects with the number of root verbs without a subject. If at least a third of root verbs do not have an overt subject, then that language was considered prodrop. This is clearly not satisfying since this ratio can greatly vary from language to language and from genre to genre. Furthermore, some languages may not be generally prodrop, but omit syntactic subjects in impersonal constructions, such as Hebrew, or be prodrop only for certain tenses.

4.5.7 Summary

Despite focusing on efficiency, our official submission obtained an average ELAS of 74.04 which was the fourth best system (out of 9 full submissions). Our improved score after training distilled models to convergence (or closer to convergence) obtained an average score of 76.14. The full breakdown of these results are shown in Table 4.16 and Table 4.17.

Our system is competitive mainly by the grace of our rule-based system which obtains an average 98.20 ELAS when used on the gold development treebanks. And for the most part its performance echoes the quality of the predicted dependencies and tags used by the

system as is seen in Table 4.13. Having a rule-based system that can perform so well on gold data means that improving the dependency predictions it is based on for a full pipeline will almost always increase ELAS scores. It also means it could be used to generate new data. Although this would be restricted to generating data for pre-existing UD treebanks. Furthermore, it could be used to highlight annotation inconsistencies in a given treebank and between different treebanks for the same language.

We also demonstrated that smaller networks can be competitive, even if in this context distillation does not perform as well as previously observed for UD parsing. And beyond that, we show that it is possible to train competitive models with less data and by doing so lowering the energy cost of training parsers. One potentially interesting result is that Tamil performs noticeably better with distillation than either the full baseline model or the small model of the same size trained normally. It has the smallest training treebank out of all the treebanks used in the shared task. The other smaller treebanks also perform better with distillation, e.g the next three smallest treebanks French, Lithuanian, and Swedish all follow this trend but the increase in performance is less pronounced. Perhaps smaller treebanks benefit from what is essentially ensemble training as it tempers a network's penchant for over-fitting.

4.6 Conclusion

In Section 4.4, we have shown that distillation can produce efficient models that are faster than the larger baselines and more accurate than equally-sized vanilla trained parsers. This results in parsers that are the fastest among modern parsers and more accurate than the next fastest variant of parsers.

However, in Section 4.5, we obtained results that do not corroborate this finding. The distilled models did not outperform their equivalently sized baseline models. This is perhaps down to a number of differences in the experimental context. This requires further analysis. The work in Section 4.5 also highlighted the high training cost of distillation when compared to training parsers normally. This means the parsers need to be used on a lot of data to make back this energy cost difference which highlights the issue of considering how parsers will be deployed when discussing efficiency costs.

One interesting nugget is that distillation seems to work better for smaller treebanks with some treebanks seeing an increase in performance over the full-size baseline. This was observed in the results for both experiments, so perhaps this version of ensemble parsing is able to offset overfitting.

There are numerous ways in which this distillation technique could be augmented to potentially retain more performance and even outperform the large baseline models, such as using *teacher annealing* introduced by Clark et al. (2019b) where the distillation process gradually secedes to standard training. Another potentially avenue is using noisy teachers if there is a larger discrepancy between the performance of the teacher and that of the student (Sau and Balasubramanian, 2016).

Beyond this, the structure of the distilled models can be altered, e.g. student models which are more shallow than the teacher models (Ba and Caruana, 2014). This technique could further improve the efficiency of models and make them more environmentally friendly by reducing the depth of the models and therefore the total number of trainable parameters.

Distillation techniques can also be easily expanded to other NLP tasks. Already attempts have been made to make BERT more wieldy by compressing the information it contains into task-specific models (Tang et al., 2019). But this can be extended to other tasks more specifically and potentially reduce the environmental impact of NLP research and deployable NLP systems.

PART III

Evaluating Parsers

Chapter 5

Dependency Displacement

Work in this chapter is based in part on published work in Anderson and Gómez-Rodríguez (2020b).

The work in this chapter focuses on explaining variance in the performance of different parsing systems by evaluating the nature of systems and data with respect to dependency displacement or edge displacement (the directed distance between a dependent and its head). In Section 5.3, we use this as the basis for investigating why certain transition-based parsers perform better or worse on certain treebanks. This is first done by comparing the performance of different algorithms for different dependency displacement bins. We then compare the inherent distribution of dependency displacements that algorithms are biased towards with the distributions observed in treebanks. In Section 5.4, we investigate why performance varies across treebanks for two parser systems by comparing the distribution of displacements observed in the training and in the test data. First we give a brief introduction to previous work investigating parsing performance and also work discussing dependency distances.

Note, because this thesis is the result of evolving research and improvements were made along the way, the definition of displacement changes in this chapter. In the initial work, it was defined as $x_{\text{head}} - x_{\text{dependent}}$ and this definition is used in Section 5.3. But in the later work, it was changed to better suit the original meaning as defined in physics, i.e. the endpoint minus the starting point. So $x_{\text{dependent}} - x_{\text{head}}$ is used in Section 5.4. The actual direction bares no impact on the results and is merely a convention (as the directionality is held constant in both analyses). Also, in Section 5.3 we refer to the Vaserstein distance (which is how we refer to it in Section 5.4) as the earth mover's distance (EMD), a popular term for it in computer science contexts.

5.1 Introduction

Evaluating the performance of NLP systems is an important task that is often done using a well-established metric or set of metrics. Error analysis often just includes cherry-picking examples that are easy to discuss but don't necessarily give a clear picture of the quality of systems. However, in the context of syntactic parsing, plenty of literature has been written discussing what factors influence parsing performance, and it is towards this discussion that this work contributes. We do so by looking at the edge displacement of nodes (the directed distance between the position of the node and its head, see Figure 5.1) and the corresponding distributions over samples. More specifically, we evaluate the inherent distributions of

transition-based algorithms and treebanks (Section 5.3) and distributions seen in training and test data of treebanks (Section 5.4). We use the Vaserstein distance (also called EMD) to measure the difference between these distributions. We then compare this with the parsing performance of different systems (or algorithms, as in Section 5.3).

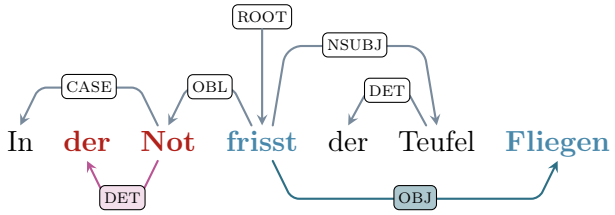


Figure 5.1: Example tree highlighting dependency displacement for two nodes. **der** at position 2 with its head **Not** at position 3 has a DET edge (in magenta) with a dependency displacement of $2-3=-1$. Similarly, **Fliegen** at position 7 with its head **frisst** at position 4 has an OBJ edge with a dependency displacement of $7-4=3$. English: *When in need the devil eats flies.* Castellano: *Por la escasez, el diablo come moscas.*

5.2 Related work

In this section we give a brief overview of previous work focused on explaining parsing performance and also focused on dependency distance.

5.2.1 Analysing parsing performance

An obvious and well-attested predictor of parsing performance is the amount of training data available, which is typically observed to be logarithmically related to parsing performance (Sagae et al., 2008; Falenska and Çetinoğlu, 2017; Strzyz et al., 2019b; Dehouck et al., 2020). The lengths of sentences have also been observed to impact parsing performance, with longer sentences being harder to parse than shorter sentences (McDonald and Nivre, 2011). In a similar vein, others have highlighted the effect dependency distance has on parsers, namely that longer dependencies tend to be harder to predict (McDonald and Nivre, 2011; Falenska et al., 2020). Edge direction entropy and word order freedom has also been shown to have a meaningful effect (Alicante et al., 2012; Rehbein et al., 2017; Gulordava and Merlo, 2015, 2016). Although this is not consistently observed across all data. Chung et al. (2010) found that for Korean this is not so strongly related to parsing performance as other features of the language such as its pro-drop tendencies. Alicante et al. (2012) only found it impacted Italian constituency parsing, but not dependency parsing. Part-of-speech bigram perplexity (Berdicevskis et al., 2018), entropy over trees (Corazza et al., 2013), the degree of non-projectivity (McDonald and Satta, 2007), and morphological complexity (Dehouck and Denis, 2018; Çöltekin, 2020) have also been presented as explanations or measurements for differences in parsing performance.

Analyses also focus on comparisons between parsing paradigms and algorithms. Transition-based parsers often appear to struggle with longer distance relations more than graph-based parsers (McDonald and Nivre, 2011; Falenska et al., 2020). However, Kulmizev et al. (2019) observed that the use of contextualised word embeddings off-set the typical issues associated with transition-based parsers. de Lhoneux et al. (2017c) investigated the performance of the same transition-based algorithm using a neural network implementation and also a classical implementation, observing the same tendency for performance to decline as dependency distance increased. Beyond this, different frameworks and annotation schemes have been found to perform differently, often related to one or more of the metrics mentioned above (Kübler et al., 2008; Matsuzaki and Tsujii, 2008; Bosco et al., 2010; Mille et al., 2012; Alicante et al., 2012; Pretkalniņa and Rituma, 2014). Differences between training and test data have also been evaluated. Zhang and Wang (2009) looked at certain metrics such as the rate of

out-of-vocabulary tokens and unseen part-of-speech trigams and observed some correlation between these and parsing performance. However, the main focus in this area is on domain shifts between training and test data. Although this issue is not unique to parsing, there have been extensive results showing that domain shift can result in very steep drops in performance if the domains are very different (Gildea, 2001; Bosco et al., 2010; Plank and van Noord, 2010; Foster, 2010). More recently, (Søgaard, 2020) proposed the ratio of tree structures in the test data that did not occur in the training data as a predictor of parsing performance, but the results presented were found to be spurious once covariants were accounted for (Anderson et al., 2021).

5.2.2 Algorithm differences

Dependency parsing, and in particular the transition-based family of parsers, has a large variety of parsing algorithms to choose from. When comparing different algorithms, empirical results on collections of corpora often show differences in accuracy that can heavily vary across different languages or treebanks, so that a given algorithm can be the best choice for one corpus while being outperformed in another.

However, despite these nontrivial patterns in accuracy variations having been observed in many experiments in the last decade, both with non-neural and neural implementations (Nivre, 2008; Ballesteros and Nivre, 2013; Chen and Manning, 2014; Fernández-González et al., 2016), very little is known about what makes an algorithm more fitting for a corpus beyond obvious facts (like non-projective algorithms being better for highly non-projective treebanks). This makes the practical choice of a particular algorithm for a language a matter of trial and error. For example, MaltOptimizer chooses between projective and non-projective algorithm according to the amount of non-projective dependencies observed in the treebank, but then the choice of a specific algorithm among projective (or non-projective) options is made by running all of the projective (or the non-projective) algorithms and choosing the one that achieves the highest accuracy (Ballesteros and Nivre, 2012).

With the proliferation of dependency treebanks for multiple languages, various papers performed comparisons including multiple transition-based parsing algorithms, where language-specific differences in performance across algorithms are apparent.

One of the first papers that included a large number of languages and algorithms is that of Nivre (2008), who found language-specific differences between the performance of Arc Standard and Arc Eager. While they hypothesized that the proportion of left arcs in a language could account for these differences, the evidence was not conclusive.

Other analyses have found differences in performances between Arc Eager and Arc Standard such as Ballesteros and Nivre (2013) or Fernández-González et al. (2016), but they provided no explanation for this phenomenon. Ballesteros and Nivre (2013) did consider the effect the dummy root placement has on different algorithms. They found that the placement is not trivial and had a noticeable effect on the performance of Arc Eager.

The differences between algorithms have also been observed with different architectural implementations, namely neural networks. Chen and Manning (2014) found that for some treebanks Arc Eager performed better whereas for others Arc Standard performed better.

5.2.3 Dependency distance

Dependency distance is hypothesised to be constrained by working memory restrictions resulting in distances being minimised (Gibson, 2000; Liu et al., 2017). This has been corroborated by numerous corpus-based analyses (Ferrer-i-Cancho, 2004; Liu, 2008, 2007; Buch-Kromann, 2006; Futrell et al., 2015; Temperley and Gildea, 2018). Although different languages appear to adhere to these restrictions to varying extents (Jiang and Liu, 2015; Gildea and Temperley, 2010). This relates to NLP parsing because if different languages or treebanks adhere to this constraint more or less than others, it could result in differences in the achievable performance of parsers. Hudson (2017) also highlighted that mean dependency

distance varies significantly between treebanks, but added that the direction of dependencies could impact parsing difficulty as well. Different syntactic traits associated with parsing difficulty have been shown to be a correlated with an increase in dependency length, e.g. free-order languages (Gulordava and Merlo, 2015) and with an increase in non-projective dependencies (Ferrer-i-Cancho and Gómez-Rodríguez, 2016; Gómez-Rodríguez and Ferrer-i-Cancho, 2017).

Gómez-Rodríguez (2017) hypothesised that transition-based parsers perform adequately because they are biased towards short dependencies. This was somewhat corroborated by Eisner and Smith (2010) who improved parser performance by imposing limits on dependency length and using dependency lengths as a feature for their system. In a response to Liu et al. (2017), Hudson (2017) highlighted that mean dependency distance varies significantly between treebanks and that the direction of dependencies could impact parsing difficulty. Liu (2010) and Jiang and Liu (2015) actually found that dependency direction analysis can be used to typologically classify languages. Hence, for our analyses we use dependency displacement which quantifies both length and direction.

5.3 Inherent Dependency Displacement Bias of Transition-Based Algorithms

A wide variety of transition-based algorithms are currently used for dependency parsers. Empirical studies have shown that performance varies across different treebanks in such a way that one algorithm outperforms another on one treebank and the reverse is true for a different treebank. There is often no discernible reason for what causes one algorithm to be more suitable for a certain treebank and less so for another. Here, we shed some light on this by introducing the concept of an algorithm's inherent dependency displacement distribution. This characterises the bias of the algorithm in terms of dependency displacement, which quantify both distance and direction of syntactic relations. We show that the similarity of an algorithm's inherent distribution to a treebank's displacement distribution is clearly correlated to the algorithm's parsing performance on that treebank, specifically with highly significant and substantial correlations for the predominant sentence lengths in Universal Dependency treebanks. We also obtain results which show a more discrete analysis of dependency displacement does not result in any meaningful correlations.

5.3.1 Data and setup

We report two levels of analysis to explain performance differences between different transition-based algorithms. The first is based on measuring the differences in performance for each dependency displacement. It transpires that this does not offer any explanation of how language-specific performance differs between algorithms. For this, we need the second analysis, based on the concept of an inherent displacement distribution of each algorithm.

We evaluate 3 projective algorithms (Arc Standard, Arc Eager and Covington) and 2 unrestricted non-projective algorithms (Covington and Swap Eager). The results for each algorithm were obtained by running Maltparser v1.91, which has the benefit of having multiple algorithms implemented of which several are projective and several are non-projective. While there are more modern systems that outperform MaltParser in accuracy, none of them provide such a range of algorithms.[1] Furthermore, as we are focused on the algorithms, the architecture of the implementation is considered a controlled variable. Also, de Lhoneux et al. (2017c) showed that the change in accuracy with respect to dependency length follows a similar trend for MaltParser and UDPipe (a neural network implementation), so it is justifiable to use MaltParser for analyses on algorithms.

[1] For example, UDPipe has one projective, one partially non-projective and one unrestricted non-projective parser.

Furthermore, MaltParser is potentially better suited for this analysis as we want to evaluate the performance of algorithms against one another, and using a more robust system that can more readily overcome the inherent biases we show in this paper would actually obscure this. For this reason too, we do not finetune the feature functions for each language as it is obvious that different features will benefit different languages more or less depending on the linguistic features of a given language. So these features are kept constant for each treebank and should be considered controlled variables for the following analyses.

Beyond these experimental considerations, MaltParser is still competitive with respect to parsing accuracy even if it is not state-of-the-art and it is very efficient, which makes it much more readily deployable when compared to large and unwieldy neural networks.

Version 2.1 of the Universal Dependency treebanks was used for all of the subsequent analysis (Nivre et al., 2017).

5.3.2 Dependency displacement

Analysis details

Similar to the analysis undertaken by (McDonald and Nivre, 2011), we investigated how parsing performance varies based on the dependency displacement. Whereas they compared a graph-based and a transition-based parser, we have compared different transition-based parsing algorithms. Also, we look at the effect dependency displacement, s_{dep}, has on the performance of the algorithms, rather than dependency distance. Here, dependency displacement is defined as:

$$s_{\text{dep}} = \text{x}_{\text{head}} - \text{x}_{\text{dependent}} \tag{5.1}$$

where x_i refers to the position of word i in a given sentence. So a right arc of length 3 would have a displacement of -3 and conversely a left arc of length 3 would have a displacement of 3. Hudson (2017) highlighted that different languages have a tendency towards being head-final (OSV or SOV), head-medial (OVS or SVO), or head-initial (VOS or VSO) and argue that the direction of dependencies cannot be ignored when analysing dependency distances. In addition, dependency direction was hypothesised by Nivre (2008) to affect how language-specific performance differs between parsers.

We evaluate the attachment precision and recall in order to have a more fine-grained analysis similar to de Lhoneux et al. (2017c).

For this analysis we used all 76 treebanks that contained a training and test set.

Results

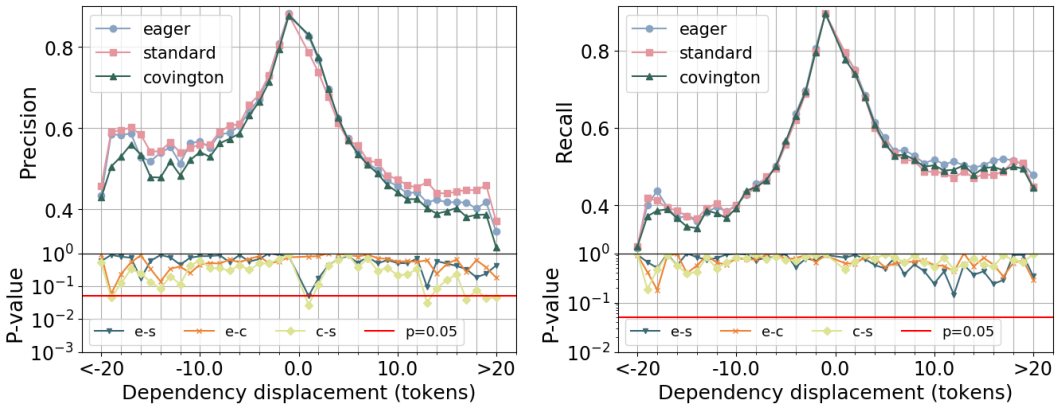
Figure 5.2 shows that this coarse analysis does not capture any statistically meaningful variation across projective algorithms with regard to attachment precision or recall. It is interesting to note that long-distant displacement to the right and to the left follow different trends for both precision and recall, with precision being higher for right arcs (negative displacement) but recall being higher for left arcs (positive displacement).

Figure 5.3 shows the corresponding results for the non-projective algorithms. We see in Figure 5.3a the only statistically meaningful difference between algorithms. Precision for left arc dependencies is higher for Swap Eager than non-projective Covington. It is also of note that the non-projective precision results are much more symmetric with regard to dependency displacement than those of the projective algorithms. These results show that there is a need for a more fine-grained analysis.

5.3.3 Comparing displacement distributions

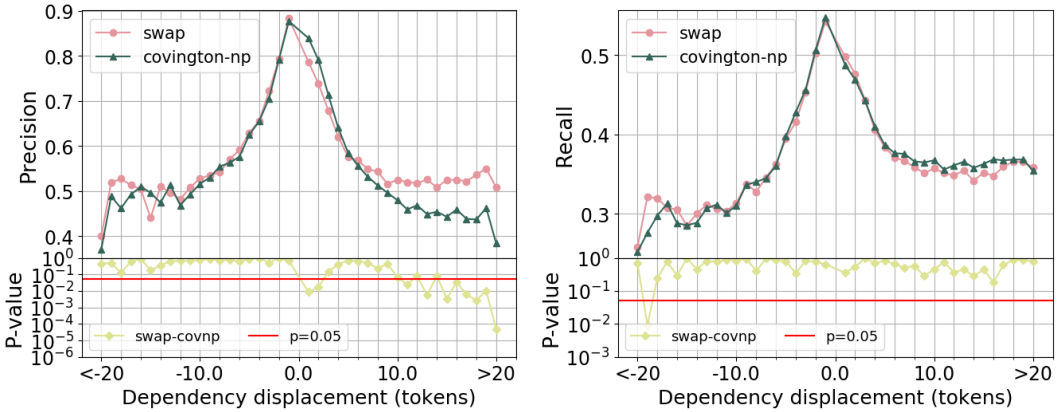
Analysis overview

In this analysis we test the hypothesis that the similarity of the dependency displacement distribution generated by the latent biases of an algorithm, as defined as its inherent displacement distribution below, and the actual distribution of a treebank can account for



(a) (b)

Figure 5.2: Attachment precision (a) and recall (b) for the three projective algorithms used: Arc Eager (eager, blue, circle), Arc Standard (standard, magenta, square), Covington (covington, green, triangle). The corresponding p-values (derived from a t-test respectively using the average precision and recall and the corresponding standard deviation across treebanks for each displacement value for each algorithm) are shown below: Arc Eager and Arc Standard (e-s); Arc Eager and Covington (e-c); and Covington and Arc Standard (c-s). No statistically significant differences are observed for any comparison with regard to attachment precision or recall.



(a) (b)

Figure 5.3: Attachment precision (a) and recall (b) for the two non-projective algorithms tested: Swap Eager (swap, magenta, circle) and non-projective Covington (covington-np, green, triangle). The corresponding p-values (derived from a t-test respectively using the average precision and recall and the corresponding standard deviation across treebanks for each displacement value for each algorithm) are shown below: Swap Eager and non-projective Covington (swap-covnp). Almost no statistically significant differences are observed for any comparison with regard to attachment precision or recall, except for left arcs (positive displacement) with regard to precision.

the difference in performance across parsing algorithms. We measure the difference between these distributions by using the Vaserstein distance - also known as the earth mover's distance (Vaserstein, 1969). The Vaserstein distance (technically the Vaserstein-1 distance) is defined as follows (Vaserstein, 1969):

$$\ell(\mu,\nu) = \inf_{\gamma\in\Gamma(\mu,\nu)} \int |x-y| d\gamma(x,y) \tag{5.2}$$

where μ and ν are probability distributions of two random variables (in our case, the variables will correspond to dependency displacements), x and y are points in the x-axis of these probability distributions (i.e. concrete values of each of the variables), $|x-y|$ is the distance between two such values, and the infimum is with respect to γ, a coupling from Γ which is the set of all joint distributions whose marginals are μ and ν.

A more grounded interpretation of the Vaserstein distance is that it gives a measurement of how much *mass* needs to be moved from each x to each y so that μ is transformed into ν (Rubner et al., 1998). As such, this metric is also known as earth mover's distance (EMD) in computing science and we use this term in this section. Ultimately it gives a measurement of how different two distributions are, with larger values indicating a greater divergence and values approaching zero indicating similar distributions. An example comparison for the German test data for sentences of length 10 to 12 tokens is shown in Figure 5.4, where Arc Standard is seen to perform worse than Arc Eager by 0.58 UAS and correspondingly has a higher EMD. We evaluate dependency displacements according to sentence-length bins, as it has been shown that dependency distance (and thus displacement) distributions change with sentence length, both in real sentences and in random models (Ferrer-i-Cancho and Liu, 2014).

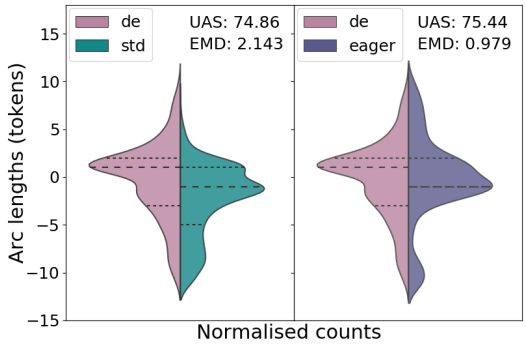


Figure 5.4: Example comparisons between the dependent displacement distribution of the German-GSD treebank (de, magenta) for sentences of length 10 to 12 in version 2.1 of the Universal Dependency treebanks and the inherent displacement distributions of two algorithms: Arc Standard (std, green) and Arc Eager (eager, purple). The corresponding UAS and EMD values are displayed.

Inherent displacement distributions

Let $P = (C, T, c_s, C_t)$ be a transition-based algorithm where C is the set of possible configurations, T the set of transitions, c_s an initialization function mapping a sentence length k to an initial configuration $c_s(k)$, and C_t a set of terminal configurations. We assume configurations in C to be of the form (D, A), where A is a set of dependency arcs built so far, and D is a state of the data structures associated with the algorithm (e.g. a stack and a buffer, for stack-based algorithms).

Then, we define the *inherent distribution* of P for sentences of length k, written $\iota_k(P)$, as the discrete probability distribution of the random variable generated by the following process:

- Start at the initial configuration $c_s(k)$.

- At each configuration, let $t_1 \dots t_q$ be the available transitions. Choose one of them randomly with probability $1/q$, and go to the resulting next configuration.
- The process ends when a terminal configuration $c_t = (D_t, A_t) \in C_t$ has been reached. Then, choose a dependency arc uniformly at random from A_t and take its displacement as the value of the random variable.

Note that the inherent distribution of an algorithm does not depend on the contents of the particular sentence being parsed in the stochastic process, but just its length. Therefore, it can be seen as a variable that describes the distribution of displacements that the algorithm is "biased" to produce, in the absence of any training data. The transition is selected using a uniform probability across all possible transitions for a given configuration as there is no underlying reason why an algorithm would select one transition over another *without* using the feature function at a given timestep. Our hypothesis is that a given language or corpus will be parsed more accurately by algorithms whose inherent distribution is closer (as measured by the EMD) to the actual observed displacements in that language or corpus.

While the inherent distribution of an algorithm for sentences of length k would be difficult to obtain analytically, especially for the algorithms that support arbitrary non-projectivity where exact inference is intractable (McDonald and Satta, 2007), in practice we will approximate it by running a number of simulations of the above stochastic process.

The above definition can be extended to corpora (or subsets of them, such as the sentence-length bins we use in this paper). Let S be a set of n sentences containing n_k sentences of length k, for a range of values of k. Then, the inherent distribution of P with respect to S is the discrete probability distribution of the random variable generated by taking a random sentence length from S (where each length k is taken with probability n_k/n), and then taking a random displacement using the process above. The inherent distribution of P with respect to a set S, denoted $\iota_S(P)$, can be approximated by running a number of simulations of the above stochastic process on all the sentences of S.

Approximating inherent distributions In order to approximate the biased distributions for each parsing algorithm P, we implemented a version of each of them so that they randomly select a transition from the set of available transitions for any given configuration. For each tree in a treebank that fell within the range of a sentence-length bin, a random tree was generated this way. This was done so as to ensure the EMD of different distributions was due to differences in the dependency displacement and to minimize other factors affecting the EMD. In other words, we wanted to obtain inherent distributions that echo the output of a parsing algorithm if it had been run normally (i.e. making predictions based on a feature function). For each treebank and each sentence length bin B in its test set, a random displacement distribution was generated 10 times to approximate $\iota_B(P)$, and their average EMD of the observed distribution of displacements in the trees of B was taken. We opted to run it 10 times individually rather than run it once with more data points as this way we can more easily evaluate the uncertainty of a given inherent distribution. The standard error for the average EMD can be observed in Figures 5.6 and 5.7. It is clear that the variation across each generated distribution is quite small and that 10 instances are enough to minimise the uncertainty of this procedure.

We split the data according to sentence length to account for the differences in arc lengths that arise from longer or shorter sentences. Optimally, we would have undertaken our analyses according to sentence length and would not have used bins, however, the statistics were too low for many treebanks in order to this. The sentence lengths bins used and their statistics can be seen in Figure 5.5.

Because of the varying difficulty of the datasets in the universal dependency collection, we have opted to compare the average EMD against δUAS, defined as the difference between the performance of an algorithm on a treebank minus the average score across algorithms

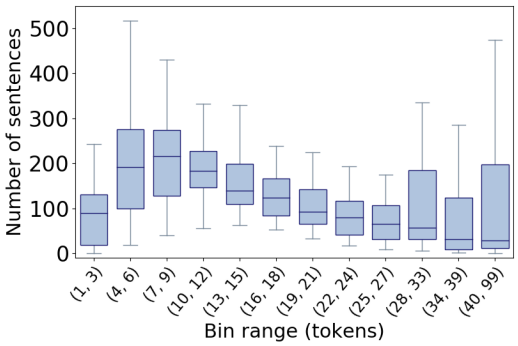


Figure 5.5: Corresponding stats for each bin used in the subsequent analysis with the average across the 26 treebanks shown and the first and third quartile limits.

for that treebank:

$$\delta\text{UAS}_{L_A} = \text{UAS}_{L_A} - \overline{\text{UAS}}_L \tag{5.3}$$

where A is the algorithm and L is the treebank.

We then compare the δUAS for an algorithm against its average EMD. We do this for 26 treebanks from version 2.1 of the Universal Dependencies treebanks. These languages were selected based on their size. We removed all languages with less than 1,000 trees in the training set and in the test set. This was necessary because if a treebank was too small then most sentence-length bins would not have enough stats to compute a meaningful EMD.

We split our analysis into projective and non-projective algorithms. The performance between projective and non-projective algorithms on certain datasets would be dominated by the percentage of non-projective arcs in the data and would potentially cloud any effect that the displacement distribution similarities might have. Nivre (2008) found a strong correlation between the percentage of non-projective dependencies and the improvement in accuracy for a non-projective parser (r=0.815, p=7.0x10^{-4}) using a much more limited dataset (CoNLL-X shared task 2006). Furthermore, the search space affects the random distributions (for example, non-projective trees in general have larger average dependency lengths than projective trees, see for example (Ferrer-i-Cancho and Gómez-Rodríguez, 2016)) and this could also be a confounding factor. So by separating projective and non-projective algorithms, we have made the search space a fixed factor.

Finally, we also compared algorithms directly. We did this by comparing ΔUAS and ΔEMD, defined as:

$$\Delta\text{UAS}_L = \text{UAS}_{L_{A_1}} - \text{UAS}_{L_{A_2}} \tag{5.4}$$

$$\Delta\text{EMD}_L = \overline{\text{EMD}}_{L_{A_1}} - \overline{\text{EMD}}_{L_{A_2}} \tag{5.5}$$

where A_1 is the first algorithm, A_2 is the second, and L is the treebank.

Results

There is a lack of meaningful correlation for both projective and non-projective parsers when looking at the displacement distributions for unbinned treebanks. The correlation and corresponding p-value for the projective algorithms were r = -0.045 and p = 6.97x10^{-1}. For the non-projective algorithms they were r = -0.252 and p = 7.17x10^{-2}. Neither result is statistically significant nor does either show any strong correlation despite that. This corroborates the findings of Ferrer-i-Cancho and Liu (2014) and further justifies our binning procedure.

Figure 5.6 shows an example plot comparing δUAS against $\overline{\text{EMD}}$ for the projective algorithms. For this bin (10-12 tokens) there is a strong negative correlation of -0.533. Hence, r^2 is 0.284, meaning that the $\overline{\text{EMD}}$ accounts for 28.4% of the variance seen in δUAS. The correlation is statistically significant ($p = 4.98 \times 10^{-7}$).

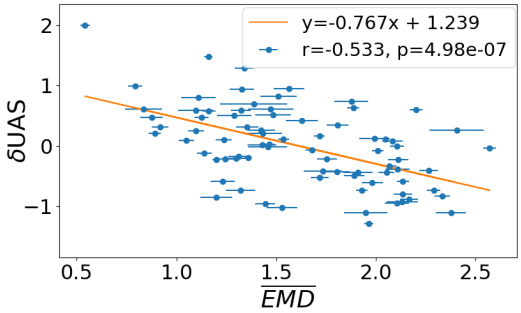


Figure 5.6: δUAS (as defined in Equation 5.3) for each algorithm against the corresponding average EMD (k=10) for projective algorithms in the 10-12 token sentence-length bin.

The corresponding results for all of the sentence-length bins can be observed in Table 5.1, where unsurprisingly the shortest sentences (lengths 1 to 3) show no correlation (they are too short for any meaningful difference between dependency displacement distributions) and the correlations start to diminish as the sentences get larger, but are still statistically meaningful until sentence lengths of 25.

Token bin	r	r^2	p
1-3	-0.060	0.004	6.09×10^{-01}
4-6	-0.401	0.161	2.75×10^{-04}
7-9	-0.503	0.253	2.74×10^{-06}
10-12	-0.533	0.284	4.98×10^{-07}
13-15	-0.526	0.277	7.49×10^{-07}
16-18	-0.514	0.264	1.47×10^{-06}
19-21	-0.402	0.161	2.68×10^{-04}
22-24	-0.304	0.093	6.78×10^{-03}
25-27	-0.202	0.041	7.65×10^{-02}
28-33	-0.072	0.005	5.29×10^{-01}
34-39	-0.034	0.001	7.70×10^{-01}
40-99	0.139	0.019	2.43×10^{-01}

Table 5.1: Full results for each sentence-length bin for projective algorithms where token bin is the sentence length range, r is the Pearson coefficient of the correlation between δUAS and the EMD of each algorithm (e.g. as shown in Figure 5.6), r^2 is the squared Pearson coefficient which gives an indication of how much variation in the data this correlation accounts for, and finally p is the p-value for a given correlation.

Figure 5.7 shows the correlation for the non-projective algorithms for the same bin as Figure 5.6. It is clear that the correlation is not as strong for the non-projective algorithms, but it is still large enough to be meaningful and is statistically significant. Table 5.2 shows the results for all of the bins used. Interestingly, the correlation does not diminish so severely for the non-projective algorithms as the sentence length increases as is the case for the projective algorithms.

Finally, focusing on direct comparisons between two algorithms, Figure 5.8 shows the comparison between Arc Standard and Arc Eager for the same bin as above. A strong negative correlation can be seen which is statistically significant. The results for the direct comparison between the three projective algorithms and the two non-projective algorithms can be seen in Figure 5.9. These comparisons yield significant results for Arc Standard when compared with Arc Eager and Covington for moderate lengthed sentences (4-25 tokens), but not for the non-projective algorithms and for Arc Eager and Covington. This suggests the previous analysis of comparing all projective and the two non-projective algorithms is a statistically more powerful means of analysing the effect of the similarity of dependency displacement distribution on algorithm performance.

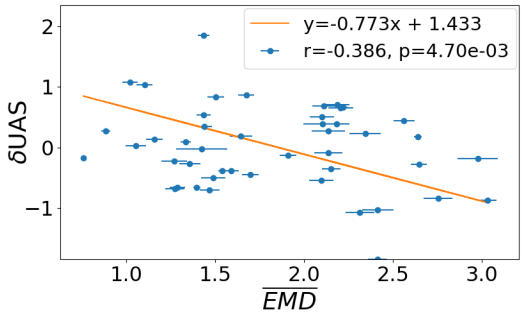


Figure 5.7: δUAS (as defined in Equation 5.3) for each algorithm against the corresponding average EMD (k=10) for non-projective algorithms in the 10-12 token sentence-length bin.

Token bin	r	r^2	p
1-3	0.001	0.000	9.92×10^{-01}
4-6	-0.243	0.059	8.32×10^{-02}
7-9	-0.327	0.107	1.79×10^{-02}
10-12	-0.386	0.149	4.70×10^{-03}
13-15	-0.344	0.118	1.25×10^{-02}
16-18	-0.364	0.133	7.90×10^{-03}
19-21	-0.344	0.118	1.26×10^{-02}
22-24	-0.350	0.122	1.11×10^{-02}
25-27	-0.349	0.122	1.11×10^{-02}
28-33	-0.347	0.121	1.17×10^{-02}
34-39	-0.354	0.125	1.01×10^{-02}
40-99	-0.298	0.089	3.97×10^{-02}

Table 5.2: Full results for each sentence-length bin for non-projective algorirthms where token bin is the sentence length range, r is the Pearson coefficient of the correlation between δUAS and the EMD of each algorithm (e.g. as shown in Figure 5.7), r^2 is the squared Pearson coefficient which gives an indication of how much variation in the data this correlation accounts for, and finally p is the p-value for a given correlation.

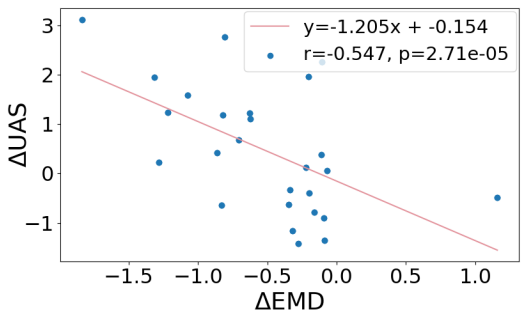


Figure 5.8: ΔUAS as defined in Equation 5.4 against ΔEMD as defined in Equation 5.5 comparing Arc Eager and Arc Standard (10-12 token sentence-length bin).

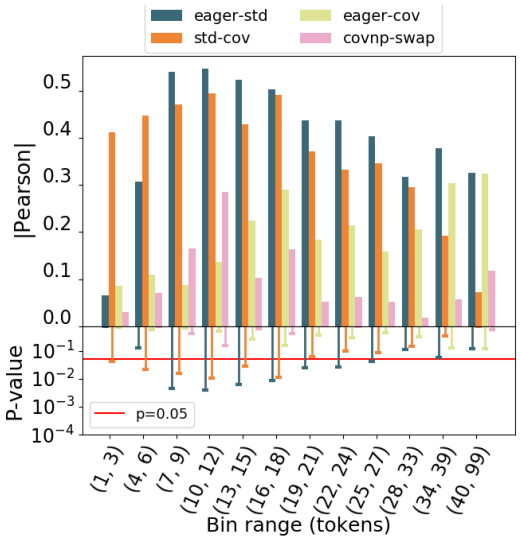


Figure 5.9: Absolute Pearson coefficients and the corresponding p-values from comparisons between each pair of projective algorithms and the two non-projective algorithms: Arc Eager and Arc Standard (eager-std, blue); Arc Eager and Covington (eager-cov, yellow); Arc Standard and Covington (std-cov, orange); and non-projective Covington and Swap Eager (covnp-swap, magenta). Statistically relevant comparisons can be seen between Arc Standard and the other projective algorithms for mid-range tree lengths.

5.3.4 Discussion

A coarse analysis focusing on dependency displacements individually does not show a significant difference in performance across algorithms except with regards to the attachment precision for non-projective algorithms. However, the difference between an algorithm's biased latent dependency displacement distribution and the target treebank being parsed is correlated with the performance of the algorithm for that treebank.

The obtained correlations are statistically significant for the sentence lengths that comprise most of the sentences found in actual corpora, both when analyzing projective and non-projective algorithms. In the case of projective algorithms, this factor accounts for more than 25% of the variance in UAS across algorithms. This is a remarkable proportion given the complexity in explaining how treebank-specific accuracy differs between algorithms and the variety of factors involved.

In fact, to the best of our knowledge, this is the first study in which these differences are studied quantitatively between transition-based algorithms of the same search space, thus casting light on a question that has been open since the introduction of the first transition-based parsers in the early 2000s.

It is worth noting that the effect of displacement distributions on parsing accuracy is independent of the effect of transition sequence length. It has been hypothesized that short transition sequences reduce error propagation (Fernández-González and Gómez-Rodríguez, 2018), but this effect does not help when comparing the relative performance of algorithms on different treebanks, as we do here: for example, the Arc Standard and Arc Eager algorithms produce transition sequences of identical lengths, independently on the syntactic structures found (they always need exactly $2n$ transitions for a sentence of length n). However, as we have seen, their inherent displacement distributions are different and can be used to explain their suitability to different treebanks.

The insights provided in this paper could be useful to guide parsing algorithm design: since algorithms tend to be more accurate on corpora that are closer to their inherent distribution, a potential avenue for designing better transition-based parsing algorithms is to try to make their inherent distribution match that of human languages more closely.

To further validate the hypothesis investigated here, it would be interesting to generate artificial treebanks in such a way so as to create a spread of arc distributions so we can control the EMD range.

Beyond the explicit findings of this study, it is interesting to observe that linguistic considerations can have an impact in natural language processing systems and more analyses like this, such as considering what makes certain languages harder to model than others, should hopefully prove to be useful in the future (Mielke et al., 2019).

5.3.5 Summary

We have introduced the concept of an algorithm's inherent displacement distribution, which captures the algorithm's bias towards implicitly preferring certain dependency lengths and directions to others. We have shown that given a treebank, the similarity between each transition-based algorithm's inherent dependency displacement distribution and the treebank's distribution is a strongly correlated to the corresponding algorithm's performance on that treebank.

5.4 Edge Displacement Vaserstein Distance

We further contribute to the ongoing discussion about parsing performance in NLP by introducing a measurement that evaluates the differences between the distributions of edge displacement (the directed distance of edges) seen in training and test data. We hypothesise that this measurement will be related to differences observed in parsing performance across treebanks. We motivate this by building upon previous work. We then attempt to falsify this hypothesis by using a number of statistical methods. We establish that there is a statistical correlation between this measurement and parsing performance even when controlling for potential covariants. We then use this to establish a sampling technique that gives us an adversarial and complementary split. This gives an idea of the lower and upper bounds of parsing systems for a given treebank in lieu of freshly sampled data.

Hypothesis We postulate that the differences in edge dependency displacement as measured by the Vaserstein distance (formally introduced in Section 5.4.1) are related to the performance of parsers (as defined by the labelled attachment score). We use a number of methods in an attempt to falsify this hypothesis and conclude that based on the data and systems used in this analysis, it cannot be refuted.

Utility We suggest using the observed correlation of Vaserstein distances between edge displacement distributions and parsing performance to guide a sampling method to create adversarial and complementary splits better suited for evaluating parsers.

The work presented here can be considered an extension of the work in the section above, where we use a method based on edge displacement distributions to compare differences between training and test data to attempt to explain variation in parsing performance across different treebanks.

5.4.1 Methodology

We here introduce the core principles behind the measurement, and we give the details of the parsing systems and data used in our analysis.

Edge displacement Vaserstein distance

As in the previous study, we use edge displacement instead of distance as this gives us a measurement that encodes both distance and direction. Fundamentally it is the signed distance of a node with respect to its head. We alter the definition from the previous section, so that it better resembles the standard definition of physical displacement, i.e. the endpoint minus the starting point:

$$s_{\text{edge}} = x_{\text{node}} - x_{\text{head}} \tag{5.6}$$

Then for a given treebank the edge displacement for each node is measured, excluding the root node and its displacement with respect to the dummy root as position 0. The distribution of edge displacements is then normalised such that it takes the form of a probability distribution. In this way, a probability distribution over displacements is obtained for the training treebank and test treebank for each dataset. We then use these two probability distributions to calculute the Vaserstein distance, thus obtaining the edge displacement Vaserstein distance (EDV) for a given dataset.

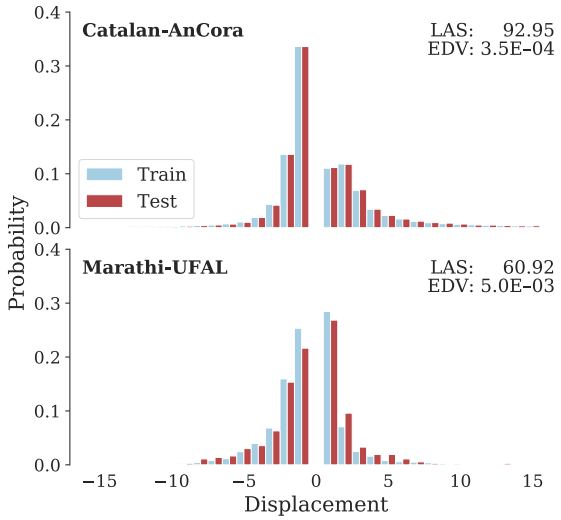


Figure 5.10: Example displacement distributions of the training and test data for Catalan-AnCora (top) and Marathi-UFAL (bottom) which exhibit the smallest and largest measured EDV values in UD v2.6. While both EDV values are small, there is an order of magnitude difference between them. LAS is shown for UDPipe 2.0.

Example distributions are shown for two treebanks from the Universal Dependency (UD) v2.6 treebanks in Figure 5.10. As can be seen, Catalan-AnCora has very similar distributions for its training and test data, which is reflected in a small EDV of 3×10^{-4}. Marathi-UFAL is also shown where differences between the two sets can be clearly seen despite the distributions following similar trends. This still results in a small EDV of 5×10^{-3}, but it is an order of magnitude greater than that observed for Catalan-AnCora. These two treebanks show the highest EDV (Marathi-UFAL) and the lowest (Catalan-AnCora) and so show the range of EDV values observed in the data (the mean EDV observed in UD v2.6 is $1.40(0.85) \times 10^{-3}$, and $1.35(0.87) \times 10^{-3}$ for UD v2.5) . Despite the values of EDV both being fairly small, there is a large difference in performance seen for these two treebanks with Catalan-AnCora achieving a labelled attachment score (LAS) of 92.95 when using UDPipe 2.0, and Marathi-UFAL only achieving 60.92. There are clearly other contributing factors relating to the difference in performance between these two treebanks (not least training data size, as Marathi-UFAL only has 373 training instances whereas Catalan-AnCora has 13,123) which we have discussed above and which we take into consideration in our analysis discussed below.

Parser systems

We used two neural based parsers: version 1.2.1-devel (1.2) of UDPipe, and version 2.0 (Straka and Straková, 2017; Straka, 2018). For UDPipe 1.2 we use models 2.5[2] and for UDPipe 2.0 we use models 2.6.[3] We opted to use these systems as the models have been optimised for their respective UD treebank and UDPipe 1.2 is a transition-based system while UDPipe 2.0 is a graph-based system, thus allowing us to evaluate EDV for different

[2] https://lindat.mff.cuni.cz/repository/xmlui/handle/11234/1-3131
[3] https://lindat.mff.cuni.cz/services/udpipe/

parser systems. Furthermore, UDPipe 1.2 came 8th out of 33 at the CoNLL 2017 shared task and was used as the baseline model for comparison of systems submitted to the CoNLL 2018 shared task, where it came 18th out of 26 with respect to average LAS. For its part, UDPipe 2.0 was one of the top performing parsers of the 2018 shared task, tied for the 3rd place (Zeman et al., 2017, 2018). An earlier version of UDPipe 2.0 was also one of the leading systems at the SIGMORPHON 2019 shared task, and the winner of EvaLatin 2020 (McCarthy et al., 2019; Sprugnoli et al., 2020).

Both systems include tokenisation and sentence segmentation capabilities, but we fed gold tokenised data to the systems as we are interested in the impact of EDV on parsing specifically and not how it relates to these preliminary tasks. When running the systems, we opted to run the taggers when parsing so as to use the systems close to how they were intended to be used, even though we are not interested in the tagging performance (of UPOS and mfeats). This results in using predicted tags at runtime.

UDPipe 1.2 is a basic feed-forward neural transition-based parser which uses a simple feature function as input for each timestep (Chen and Manning, 2014; Straka et al., 2015). We used models 2.5 which were pre-trained on UD v2.5 treebanks, resulting in 94 parsers on separate treebanks. Each model is optimised for each treebank which includes the type of algorithm and oracle used. Details of the system can be found in Straka et al. (2015) and Straka et al. (2016).

UDPipe 2.0 is based on the graph-based biaffine parser of Dozat and Manning (2017) where the hidden representations of tokens from BiLSTM layers are mapped into two separate perceptron layers, considered representations of the tokens as a head and as a dependent, which are combined using a biaffine attention mechanism, resulting in a probability distribution over all other tokens in a sentence indicating the probability that any given token is its head. A well-formed tree is then enforced using the Chu–Liu/Edmonds' algorithm (Chu and Liu, 1965; Edmonds, 1967). We could not run Czech-PDT, Hindi-HDTB, German-HDT, and Russian-SynTagRus as the website had issues with large files, so we ended up with results from 90 models. Note that while the treebanks used for UDPipe 1.2 and 2.0 are very similar, they are not exactly the same. There are few differences in the actual treebanks included and there are also differences within given treebanks between iterations of UD releases.

Data

We used UD treebanks for our analysis (as such, we lay no claim to any results that span different frameworks). We used the sets of treebanks that correspond to the parser models we used for each system, namely UD v2.6 with UDPipe 2.0 and UD v2.5 for UDPipe 1.2. We also used UD v2.7 to extend our analysis beyond the pretained model for a evaluation of the linear regression model using unseen data. We picked treebanks that had no UDPipe 1.2 model but contained both training and test data and which contained at least 100 sentences in the training data. We also used UD v2.7 for a proof of concept for using EDV to guide sampling for a more robust evaluation procedure for parsers. This resulted in 94 treebanks for UDPipe 1.2, 90 treebanks for UDPipe 2.0, 11 treebanks for evaluating the UDPipe 1.2 linear model, and 105 treebanks for the sampling work.

Statistical methods

All statistical analysis was undertaken using the Pingouin Python library version 0.3.8 (Vallat, 2018).

Correlation coefficients We evaluate the impact variables have on parsing performance by measuring their correlation coefficients with respect to LAS. We use a non-parametric correlation coefficient in the form of Spearman's ρ, which measures the correlation between variables and assesses the monotonic relationship between them. We do not use Pearson's r as the data being analysed do not strictly adhere to bivariate probability distributions

and the sample sizes are small enough that this can affect the measurement's sensitivity. Further, Pearson's r is less robust with respect to outliers. For each coefficient, we report the correlations and the corresponding p-value. For the main correlation results, we include the upper and lower bounds of the 95% confidence interval, the coefficient squared (a measure of the proportion of explained variance), the adjusted coefficient which somewhat tempers the coefficient's bias, and the power of the analysis. For p-values, we report the exact value unless the value is less than 0.001, following common practice (American Psychological Association., 2010).

Partial correlations We make use of partial correlations to evaluate the impact of covariants. This allows us to remove the impact of variables that are correlated with the control variable and the target variable, so as to avoid situations where a measurement seemingly explains X variance in the data but in reality it is merely a measurement of one or more basic variables.

Background removal Here we take a standard method found in physics used to remove known background functions from data, e.g. removing the spectra associated with amorphous radiators from those associated with lattice-structure radiators to obtain enhanced spectra, i.e. without noise (Timm, 1969). Here we consider the variations associated with covariants as similar background data to be removed, so as to observe if there is any variation associated with EDV. Similar to partial correlations, removing the background signal of a potential covariant allows us to visually evaluate the specific impact a variable of interest has on the target variable. This involves fitting the control data and the target (e.g. the size of training data and LAS) and then dividing the target variable by the predicted values from this fit. This *normalised* data is then used to fit a second potential covariant which too is used to divide the normalised target variable values. This can be repeated for any number of covariants. Ultimately a normalised version of the target variable is left and the control target of interest (e.g. EDV) is evaluated against these values and if a trend is still observed, it is evidence that this variable has an impact on the target variable even with the variance associated with these covariants removed. This technique ultimately acts as a way of tempering correlations we calculate and gives us a means of disentangling contributions that might not be caught by partial correlation calculations.

Linear regression The preceding methods allow us to hone in on the impact of a given variable, but with linear regression we can fit models to the data with more than one variable. This allows us to evaluate the impact certain variables have when used with other covariants. For linear regression models we report the adjusted R^2 (the square of the residuals) as a measurement of the proportion of explained variance, which it equals when the residual mean is normalised so as to equal zero (as is the case in this analysis). In addition, we report the relative importance of each variable and the corresponding p-values (Sen et al., 1981; Groemping, 2006).

Sentence length binning Ferrer-i-Cancho and Liu (2014) highlighted the impact mixing sentence lengths can have on treebanks analyses and Anderson and Gómez-Rodríguez (2020b) observed sentence-length dependencies when evaluating edge displacement distributions of treebanks and the inherent distributions of transition-based parsers. Considering this potential impact, we also undertake a sentence-length binned analysis. This simply entails constructing samples of each treebank based on the length of the sentences. We take bins ranging from 3 tokens to 30 tokens, as any shorter and the EDV has little meaning (i.e. with 2 tokens, there can only be one edge which can either be -1 or 1) and any longer and the number of instances in a given bin for a given treebank is too small to obtain a meaningful measurement. Note that parsers were trained on the full data and the binning procedure is undertaken solely at the analysis stage. Figure 5.11 shows the EDV calculated between training and test data for each sentence length bin for UD v2.6 (the corresponding data

for UD v2.5 is shown in Figure 5.35 in Appendix 5.A.2). It is clear that EDV does vary based on sentence length, but it remains to be seen whether that variation has an impact on parsing performance.

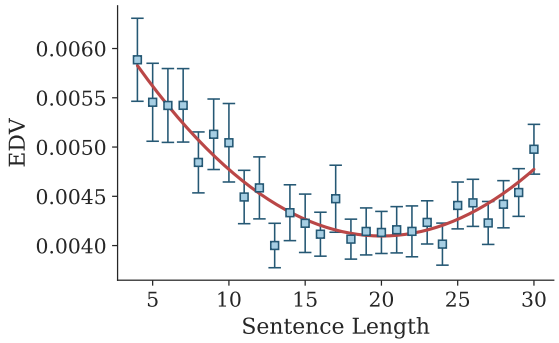


Figure 5.11: EDV between sub-samples of the training and test data binned by sentence length for UD v2.6 (111 treebanks).

Variables assessed Beyond assessing EDV and how it correlates to parsing performance (as given by LAS) we look at a number of variables which are potential covariants. First we look at the size of the training data (measured both in tokens and sentences), which as described above has been shown to correlate to parsing performance and could feasibly impact EDV measurements, i.e. larger treebanks allow for a more *accurate* representation of a language's true underlying distributions of edge displacements so deviations with respect to the test data could be minimised, and vice versa: if the sample is too small, it could be some random sample at the fringes of what would be a standard distribution for a given language. Similarly, we also consider the number of tokens and sentences in the test data. We also look at the mean sentence length of the test data, $\langle L_{\text{test}} \rangle$, as this theoretically puts a limit on the potential distribution of edge displacements and has been observed to impact parsing performance (i.e. longer sentences are harder to parse than shorter ones). For the sake of completeness, we also look at the mean length of the training data, $\langle L_{\text{train}} \rangle$. Finally, we look at the Vaserstein distance between the training and test distributions of sentence lengths (SLV) because it is feasible that EDV merely vaguely measures differences with respect to sentence length.

5.4.2 Analysis and results

In this section we describe the analysis in detail and discuss the results we obtained.

Evaluating normality

Here we justify the use of Spearman's ρ for the following analysis. Figure 5.12 shows the distributions of the variables of interest in our analysis (as described in Section 5.4.1) for UD v2.6 (the corresponding distributions for UD v2.5 are shown in Figure 5.34 in Appendix 5.A.2). Visually, it is clear that only $\langle L_{\text{test}} \rangle$ could be sampled from a normal distribution.

To thoroughly evaluate the variables for normality, we use the Shapiro-Wilk test (Shapiro and Wilk, 1965) as it is a higher power test compared to the alternatives, making it the most suitable for our fairly small sample size (Yap and Sim, 2011). The values from the tests (W) and the corresponding p-values (where the null hypothesis is that the sample *is* from a normal distribution) are shown in Table 5.3 for both UD v2.5 (top) and UD v2.6 (bottom). A smaller W indicates that a sample is not drawn from a normal distribution, but the more informative metric here is the p-value (as W is non-linear and difficult to interpret). Basically, larger p-values mean we cannot reject the null hypothesis that the sample is drawn from a normal distribution. Only $\langle L_{\text{test}} \rangle$ has a large p-value and does so for both datasets (0.121 for UD v2.5 and 0.402 for UD v2.6). The left most column of Table 5.3

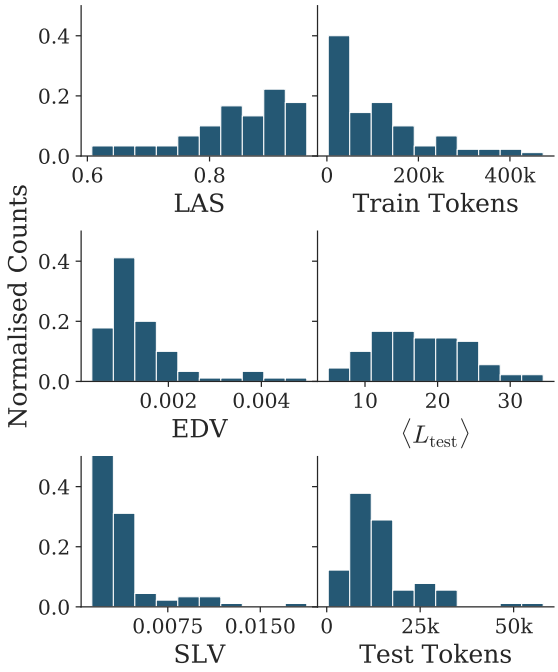


Figure 5.12: Distributions of the variables of interest in UD v2.6 (90 treebanks) in order to evaluate whether they are sampled from normal distributions.

Variable	**W**	**p-value**	**Normal**
LAS	0.920	<0.001	False
Train Tokens	0.418	<0.001	False
EDV	0.785	<0.001	False
$\langle L_{\text{test}} \rangle$	0.978	0.121	**True**
SLV	0.686	<0.001	False
Test Tokens	0.350	<0.001	False
LAS	0.894	<0.001	False
Train Tokens	0.851	<0.001	False
EDV	0.761	<0.001	False
$\langle L_{\text{test}} \rangle$	0.985	0.402	**True**
SLV	0.665	<0.001	False
Test Tokens	0.825	<0.001	False

Table 5.3: Shapiro-Wilk tests to evaluate if samples are drawn from normal distributions for UD v2.5 (top) and UD v2.6 (bottom). Only the $\langle L_{\text{test}} \rangle$ test has values for which the null hypothesis (i.e. normal distribution) cannot be rejected under any reasonable thresholds.

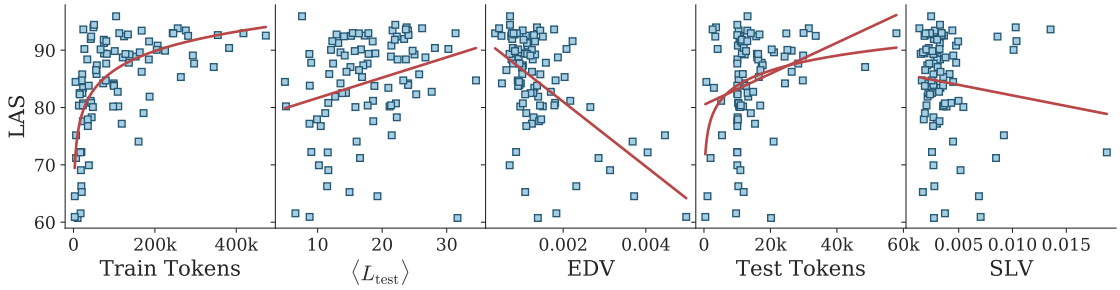


Figure 5.13: Visualisation of LAS (for UDPipe 2.0 and UD v2.6) with respect to variables of interest with fits shown in red to highlight whether the data appears correlated or not.

shows the result of the test based on the ever arbitrary distinction of significance, i.e. p-value < 0.05. We are not particularly interested if one variable is or is not normally distributed, the important result here is that most variables inluding the control variable of interest (EDV) and the target variable (LAS) quite definitively do not follow normal distributions. This along with the other considerations mentioned in Section 5.4.1 thoroughly justifies the use of Spearman's ρ. Further, it is useful that this coefficient doesn't specifically evaluate the linearity of relationships because not all variables assessed here are linearly related to parsing difficulty, but *are* monotonically related.

Correlation coefficients

Here we evaluate basic coefficients between the control variables and LAS and also between the potential covariants and EDV.

Basic coefficients

Figure 5.13 shows LAS against the control variables of interest for UDPipe 2.0 (the corresponding visualisation for UDPipe 1.2 is shown in Figure 5.37 in Appendix 5.A.2). In the first subplot, it is fairly clear that LAS increases logarithmically with respect to the number of tokens in the training data, which corroborates the findings discussed above in Section 5.2. It appears that the number of tokens in the test data is not associated with parsing performance for UDPipe 2.0, however, there is a potentially logarithmic relationship seen for UDPipe 1.2, but that could easily be down to a few serendipitously placed outliers. $\langle L_{\text{test}} \rangle$ is loosely linearly related to LAS, but EDV seems like it is more strongly linearly related. SLV doesn't seem to be related to LAS, but there are a few clusters which upset the fitting procedure that should not affect the calculation of the corresponding Spearman ρ for this relation. Note that we do not visualise all variables for the sake of space and to avoid redundancy, i.e. the number of training tokens is more strongly correlated to parsing performance than the number of training sentences (as seen in Table 5.4).

Parser	Variable	ρ	p-value
UDPipe 1.2	Train Tokens	0.660	<0.001
	Train Trees	0.535	<0.001
	$\langle L_{\text{train}} \rangle$	0.376	<0.001
	Test Tokens	0.433	<0.001
	Test Trees	0.208	0.045
	$\langle L_{\text{test}} \rangle$	0.351	0.001
	SLV	-0.191	0.065
	EDV	-0.492	<0.001
UDPipe 2.0	Train Tokens	0.605	<0.001
	Train Trees	0.467	<0.001
	$\langle L_{\text{train}} \rangle$	0.323	0.002
	Test Tokens	0.309	0.003
	Test Trees	0.073	0.496
	$\langle L_{\text{test}} \rangle$	0.309	0.003
	SLV	-0.086	0.422
	EDV	-0.466	<0.001

Table 5.4: Spearman's ρ for correlations between variables of interest and LAS.

Table 5.4 shows the corresponding Spearman ρ values for the data shown in Figures 5.13 and 5.37 and the remaining variables mentioned above in Section 5.4.1, i.e. control variables related to LAS. First, we want to note that measuring data in tokens rather than sentences results in stronger correlations for both test and training and for both parsers (with the number of test instances not even being correlated to LAS for UDPipe 2.0). Based on this, we use the number of tokens in the training and test data from here on in. Also, the number of training tokens is the variable most strongly correlated with parsing performance, but the next strongest for both systems (excluding the number of training sentences) is actually

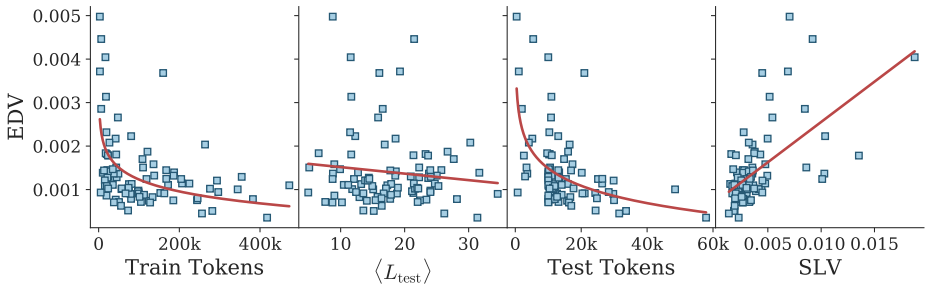


Figure 5.14: Visualisation of EDV (for UD v2.6) with respect to variables of interest with fits shown in red to highlight whether the data appears correlated or not.

EDV. SLV is not correlated at all for UDPipe 2.0 and only weakly so for UDPipe 1.2, with a p-value higher than any arbitrary threshold of significance.

Next we investigate how the variables most strongly correlated to LAS correlate with one another, i.e. we check for potential covariants. Figure 5.14 shows how pertinent variables relate to EDV. Clearly, the number of tokens in the training data and the test data are strongly related and, as one would expect, SLV looks related (confirmed by the actual correlation coefficient of 0.549 with a p-value less than 0.001 as seen in Table 5.5). However, as SLV is not correlated to parsing performance, it is not necessary to consider it when evaluating EDV with respect to LAS. It seems like $\langle L_{\text{test}} \rangle$ is not clearly related to EDV despite our expectations that it would be.

Parser	**Variables**	ρ	**p-value**
UDPipe 1.2	Train Tokens — EDV	-0.480	<0.001
	$\langle L_{\text{test}} \rangle$— EDV	-0.080	0.443
	$\langle L_{\text{train}} \rangle$— EDV	-0.089	0.393
	Test Tokens — EDV	-0.523	<0.001
	SLV — EDV	0.617	<0.001
	Test — Train (Tokens)	0.772	<0.001
	$\langle L_{\text{test}} \rangle$— Train Tokens	0.149	0.153
UDPipe 2.0	Train Tokens — EDV	-0.424	<0.001
	$\langle L_{\text{test}} \rangle$— EDV	-0.025	0.817
	$\langle L_{\text{train}} \rangle$— EDV	-0.023	0.833
	Test Tokens — EDV	-0.446	<0.001
	SLV — EDV	0.549	<0.001
	Test — Train (Tokens)	0.659	<0.001
	$\langle L_{\text{test}} \rangle$— Train Tokens	0.096	0.370

Table 5.5: Spearman's ρ for different pairs of variables.

The corresponding correlations are found in Table 5.5 alongside correlations between other variables as well. The correlations clearly corroborate the trends observed in Figure 5.14. $\langle L_{\text{train}} \rangle$ is not shown in Figure 5.14 but it behaves similarly to $\langle L_{\text{test}} \rangle$, closely echoing the measured correlations between $\langle L_{\text{test}} \rangle$ and EDV for both systems. We also show the correlation between the number of training tokens and test tokens as typically the amount of data for both are linked (i.e. it is not particularly common for a treebank to have a huge training set but a tiny test set, although the opposite does occur, eg. Kazakh-KTB). For both sets of data the correlations are high (0.772 for UD v2.5 and 0.659 for UD v2.6) both with p-values below 0.001. We assume, therefore, that these measurements loosely capture the same aspect of treebanks and use the number of training tokens as the best option: it is more strongly correlated to LAS by a large amount and is similarly correlated to EDV if slightly less so than the number of test tokens. We further justify this choice in the "Background removal" section below. Lastly, we show the correlation of $\langle L_{\text{test}} \rangle$and the number of training tokens as it has been noted that smaller treebanks (especially very

low-resource treebanks) not only have less training instances but also sentences tend to be shorter (Dehouck and Gómez-Rodríguez, 2020). However, we don't find any correlation in these datasets, presumably because this issue is not prevalent once a certain threshold of data size is reached.

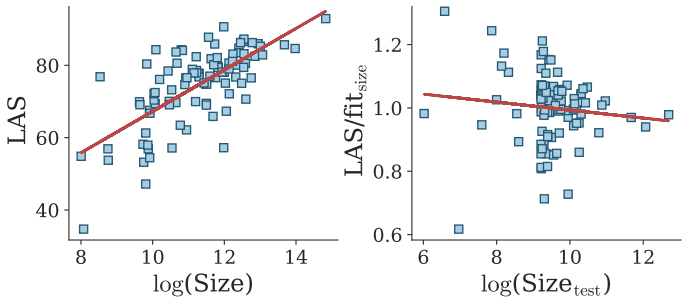


Figure 5.15: Background removing method used to evaluate whether the number of test tokens carries additional information with respect to the number of training tokens for UDPipe 1.2 and UD v2.5. Correlation between the number of test tokens and LAS is 0.433 (p-value<0.001) and that between the number of test tokens and the normalised LAS (right plot) is -0.123 (p-value=0.236).

Background removal

As described above, we removed the background signal associated with other variables to evaluate the independent relationship of certain variables. First, we evaluated whether the number of test tokens actually captured a different aspect of the treebanks with respect to parsing performance. Figure 5.15 shows this process for UDPipe 1.2, where the first plot shows LAS against the number of training tokens and the second plot shows the normalised LAS (LAS / fit from first plot) against test tokens.

Parser	Covariant(s)	ρ	CI95%	ρ^2	Adjusted ρ^2	p-value	power
UDPipe 1.2	None	-0.492	[-0.63 -0.32]	0.242	0.225	<0.001	0.999
	Train Tokens	-0.364	[-0.53 -0.17]	0.132	0.113	<0.001	0.955
	Train Tokens, $\langle L_{\text{test}} \rangle$	-0.333	[-0.50 -0.14]	0.111	0.091	0.001	0.912
UDPipe 2.0	None	-0.466	[-0.61 -0.29]	0.217	0.199	<0.001	0.997
	Train Tokens	-0.348	[-0.52 -0.15]	0.121	0.101	0.001	0.925
	Train Tokens, $\langle L_{\text{test}} \rangle$	-0.346	[-0.52 -0.15]	0.119	0.099	0.001	0.922

Table 5.6: Partial coefficients (except for rows with None in Covariant(s) column) for EDV with respect to LAS for UDPipe 1.2 and UD v2.5 (top) and for UDPipe 2.0 and UD v2.6 (bottom). Shown is the coefficient itself (ρ), the 95% confidence interval (CI95%), ρ^2 as an indication of the proportion of explained variance, the adjusted ρ^2 (Adjusted ρ^2) as a less biased version of ρ^2, the corresponding p-values, and the achieved power of the test (power).

We show this process for UDPipe 1.2 rather than 2.0 which we have used for the visual representations in the main body thus far (the corresponding plot for UDPipe 2.0 is shown in Figure 5.36 in Appendix 5.A.2) as the visual relationship observed for UDPipe 1.2 between the number of test tokens and LAS was much more convincing than for UDPipe 2.0 and the correlation reported in Table 5.5 was higher for UDPipe 1.2. It is clear that once removing the signal associated with the number of training tokens, the signal associated with the number of test tokens disappears. This is backed up by the correlations observed for the number of test tokens and LAS (0.433, p-value<0.001) disappearing when comparing the number of training tokens to the normalised LAS with a correlation of -0.123 (p-value=0.236).

We note here that when looking at the partial coefficient for the number of test tokens for UDPipe 1.2 when using the number of training tokens as a covariant, we obtain a coefficient of -0.325 (p-value=0.001) which is not particularly meaningful and highlights the fragility of correlation coefficients. In fact, the reversal of the sign is indicative of multicollinearity,

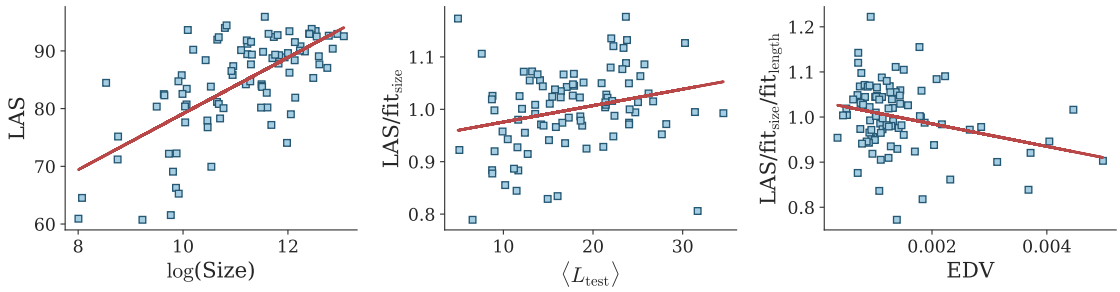


Figure 5.16: Background removal method to evaluate whether a correlation is observed between EDV and LAS (for UDPipe 2.0 and UD v2.6) after removing the variation associated with the training test size and $\langle L_{\text{test}} \rangle$. The correlation between EDV and LAS is -0.466 (p-value<0.001), the correlation between EDV and the LAS normalised by the variance associated with number of tokens in training data is -0.222 (pvalue=0.036), and the correlation for the fully normalised LAS (removing the variance associated with $\langle L_{\text{test}} \rangle$) is -0.283 (pvalue=0.007).

Parser	Variables	Adj. R^2	Relative Importance	p-values
UDPipe 1.2	logTrain Tokens	0.475	100.0	<0.001
	logTrain Tokens, $\langle L_{\text{test}} \rangle$	0.503	87.8, 12.2	<0.001, 0.015
	logTrain Tokens, EDV	0.567	55.7, 44.3	<0.001, <0.001
	logTrain Tokens, $\langle L_{\text{test}} \rangle$, EDV	0.589	50.8, 8.6, 40.6	<0.001, 0.018, <0.001
UDPipe 2.0	logTrain Tokens	0.434	100.0	<0.001
	logTrain Tokens, $\langle L_{\text{test}} \rangle$	0.468	88.3, 11.7	<0.001, 0.012
	logTrain Tokens, EDV	0.494	61.4, 38.6	<0.001, 0.001
	logTrain Tokens, $\langle L_{\text{test}} \rangle$, EDV	0.522	56.0, 9.1, 35.0	<0.001, 0.015, 0.001

Table 5.7: Statistics associated with linear regression models using combinations of log size, EDV, and $\langle L_{\text{test}} \rangle$ as predictors. We report the adjusted R^2 (Adj. R^2) scores for linear regression fits as a less biased indication of the proportion of explained variance and report the percentage of relative importance of each predictor along with the corresponding p-values.

exactly what we anticipated these variables to be (Farrar and Glauber, 1967). For UDPipe the same partial correlation is −0.045 (p-value=0.671) and so it is even clearer for this system.

We next use this technique to evaluate the relationship observed between EDV and LAS. In Figure 5.16 we show the fit of LAS against the number of training tokens (leftmost plot), and then the first normalised LAS against $\langle L_{\text{test}} \rangle$ (middle plot), and the final normalised LAS against EDV (rightmost plot) for UDPipe 2.0 (Figure 5.39 in Appendix 5.A.2 shows the equivalent analysis for UDPipe 1.2). We opted to include $\langle L_{\text{test}} \rangle$ even though no correlation was observed between $\langle L_{\text{test}} \rangle$ and EDV because theoretically it could impact the measurement of EDV, and if the coefficients failed to capture this, it could still impact the final analysis. However, removing the signal associated with it and the number of training tokens still results in a clear linear relationship between EDV and LAS (correlation of −0.283 with p-value=0.007). The correlation is much diminished compared to the original coefficient measured for EDV of −0.466 (Table 5.4), but it is still meaningful. The results are echoed in the analysis for UDPipe 1.2 with a correlation of −0.249 (p-value=0.015) between EDV and the final normalised LAS compared to −0.492 for the original measured coefficient (Table 5.4).

Partial coefficients

This ultimately leads us to evaluating EDV with respect to LAS using partial coefficients. The main covariant of interest is the number of tokens in the training data, which is not only the most strongly correlated variable with respect to LAS (Table 5.4) but also the second most strongly correlated variable with respect to EDV (Table 5.5). We also include $\langle L_{\text{test}} \rangle$

despite measuring no correlation with it and EDV because of the apparent impact it had in the background subtraction analysis In Table 5.6, we show the full measurement of the partial coefficients for EDV with respect to LAS for UDPipe 1.2 and 2.0 with no covariants (i.e. the standard coefficient), with the number of training tokens as the sole covariant, and with both the training tokens and $\langle L_{\text{test}} \rangle$ as covariants. As expected, when evaluating the correlation with the number of training tokens as a covariant we observe the biggest change in the measured coefficient. For UDPipe 1.2 it drops from -0.492 to -0.364 and for UDPipe 2.0 it drops from -0.466 to -0.348. We also note that despite not being correlated based on the calculated coefficients between $\langle L_{\text{test}} \rangle$ and EDV, there is still a small decrease in the partial coefficients here. This partial correlation coefficient, the most conservative, results in an adjusted ρ^2 of 0.091 for UDPipe 1.2 and 0.099 for UDPipe 2.0, which gives a less biased indication of the proportion of explained variance associated with EDV (9% for UDPipe 1.2 and 10% for UDPipe 2.0).

Multilinear regression

We then evaluated the impact EDV has in a multilinear regressive fit of the data for both systems. The results are shown in Table 5.7. We start by simply fitting a model using the log of the number of training tokens and for both systems we obtain a fit that has reasonably large adjusted R^2 (0.475 and 0.434 for UDPipe 1.2 and 2.0, respectively). We also use $\langle L_{\text{test}} \rangle$ based on the results from the "Background analysis" and "Partial coefficients" sections and see that the adjusted R^2 for the model using this and the log of training token size is slightly higher than only using the training tokens (about 0.03 for both systems). Using training tokens with EDV, however, results in a larger increase of 0.09 for UDPipe 1.2 and 0.06 for UDPipe 2.0. We also observe an increase when using EDV in addition to the other two variables which results in the largest adjusted R^2 of 0.589 and 0.522 for UDPipe 1.2 and 2.0, respectively.

It is necessary to highlight that despite reporting the adjusted R^2, it is still a biased indication of the proportion of explained variance of a model. However, it is still indicative of the quality of the model, but more importantly it allows us to evaluate the impact of EDV. We also report the relative importance percentages in Table 5.7 which show that EDV roughly carries 40% of the importance in the models it is used in for UDPipe 1.2 and about 35% for UDPipe 2.0.

Testing the model with UDPipe 1.2

As there exists a more up-to-date version of UD that contains more treebanks not used in the systems we have evaluated, we can use these new treebanks to evaluate the linear model from the "Multilinear regression" section. We select the new treebanks based solely on two criteria: that the treebanks have at least 100 training sentences (as very small treebanks tend to be very volatile with respect to performance) and that they contain pre-existing training and test sets (and potentially a development set). This resulted in 11 new treebanks. Note, Latin-LLCT fit these criteria but we opted not to use it as it contains the same sentence 356 times across the training, dev, and test data.

We trained models using UDPipe 1.2 with the general settings. This means these data points are slightly different from those used to develop the linear regression model which were all optimised for each treebank based on the algorithm and oracle used. We ran the evaluation the same as described in Section 5.4.1. We did not train models for UDPipe 2.0 as the parser is not publicly available. We then compared the LAS we obtained from these parsers and the values predicted by the linear regression model using all 3 variables as discussed in the "Multilinear regression" section. The comparisons are shown in Figure 5.17 where the predicted values are not outlandishly different for most treebanks except for those which obtained fairly low LAS. While we have not set out to develop a predictive model, this is still useful as a sanity check (if the predictions had been wildly inaccurate across

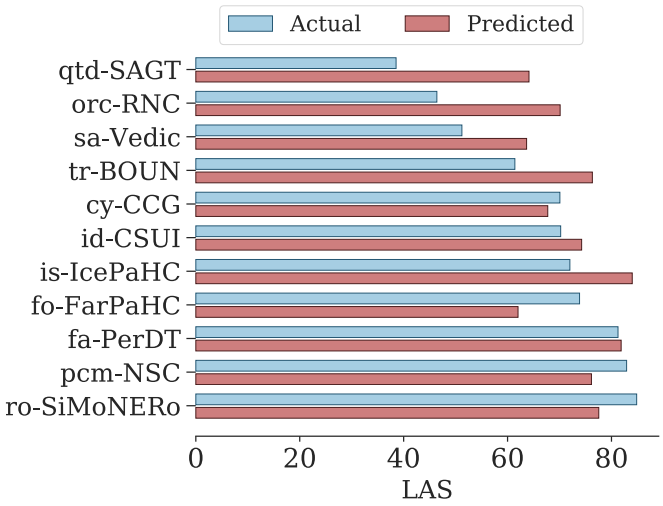


Figure 5.17: Comparison of performance of new UDPipe 1.2 models for treebanks not covered in current UDPipe 1.2 models that appear in UD v2.7 with predictions from linear model from Section 5.4.2 using the log of the number of training tokens, $\langle L_{\text{test}} \rangle$, and EDV as predictors. The mean absolute error is 11.05.

the board, then one would have to question not only the linear model but the calculated coefficients).

Sentence length binning

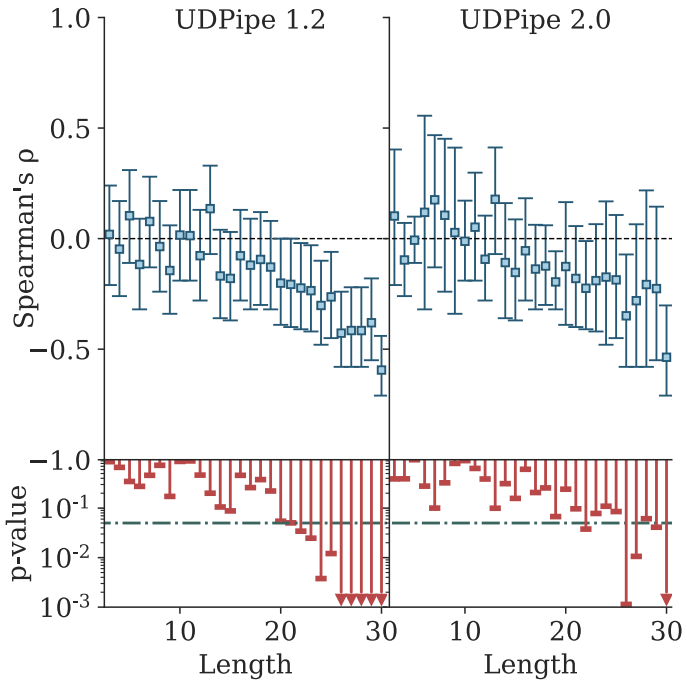


Figure 5.18: Correlation coefficients (top, blue) and their corresponding p-values (bottom, red) for UDPipe 1.2 (left) and UDPipe 2.0 for sub-samples binned with respect to sentence length. Comparison is between the EDV and LAS of each sub-sample.

Here we turn to our sentence length binning analysis. As shown above in Figures 5.11 (and 5.35 in Appendix 5.A.2), EDV does show an expected dependency on sentence length. We also would like to highlight that this dependency is hardly unique to this situation, but consideration of this is almost completely lacking in NLP. Figure 5.18 shows the partial

correlation coefficients and the corresponding p-values for each sentence length bin we evaluated in this analysis (sentence lengths of 3 to 30) for both parsers. Note that we only used the number of tokens in the training data as a covariant because for each bin $\langle L_{\text{test}} \rangle$ is constant across each treebank by design.

A clear trend can be observed where the magnitude of the correlations increases as a function of sentence length. Further the p-value decreases with respect to sentence length as well, with lower sentence lengths not showing p-values typically considered small enough to reject the null hypothesis (EDV is not related to parsing performance). We offer visualisation of the corresponding scatter plots for each bin in Figures 5.40 and 5.41 in Appendix 5.A.2 for UDPipe 1.2 and 2.0, respectively. It is clear from these plots that the correlations reported in Figure 5.18 correspond to visual trends and are not merely mathematical artefacts.

We also note that unlike other treebank analyses focusing on measurements that are likely to be related to sentence length, EDV has a clear global correlation with our target variable (e.g. in our previous analysis based on inherent dependency displacement distributions, we did not observe a global correlation). But the sentence length binning highlights that stronger signals can be found in a more fine-grained analysis.

5.4.3 EDV for evaluation

Having clearly established that EDV does correlate to parsing performance both globally when accounting for covariants in a number of ways and in a more fine-grained sentence length binning analysis, we turn to a proof of concept for a potential application of EDV in NLP: using it to inform a more linguistically motivated means of creating *adversarial* splits. We note here that large EDV values between samples for a given language likely capture a linguistic feature of that language, in that large samples that deviate to a great extent suggest that language is more syntactically volatile than others. This could be true across the board or it could be a matter of greater variety in syntactic structures in a given domain, e.g. long, anfractuous sentences are more likely be found, for better or worse, in the ramblings of literary minds as attested to by any given utterance penned by Proust while shorter sentences are more likely to found in military memos. However, while differences likely do occur based on domain, there is just as much, if not more, intra-domain variety than inter-domain, e.g. Orwell's call for simpler, clearer writing (Orwell, 2002).

We mention this here because recent work on developing adversarial splits focused on sentence lengths (Søgaard et al., 2021). This was an extension from criticism based on using standard splits, where random splits were suggested instead (Gorman and Bedrick, 2019). Together these analyses showed that standard splits and random splits are not enough to truly evaluate the brittle nature of NLP systems trained on data from a narrow set of domains. Søgaard et al. (2021) found that even when evaluating systems with adversarial splits (based on sentence length), the evaluation over-estimated the performance of the systems when compared with fresh samples. We argue that creating adversarial splits based on sentence length is only weakly linguistically motivated (i.e. the variance in sentence length could be associated with different domains, but maximising the difference between test and training set is only a very coarse approximation of differences in domain, as not all long sentences are necessarily harder for a model to handle). With this is mind, we propose using EDV to guide sampling to create adversarial and complementary splits to give an approximation of the volatility of parsing performance. As highlighted by Søgaard et al. (2021), this only offers us a clearer picture of the generalisability of models based on the data available, which often overestimates the quality of models. However, in lieu of fresh data, this offers us a clear path to a more robust evaluation.

Sampling

We sample data in such a way so as to minimise EDV and maximise EDV, in order to give certain empirical limits of performance for a given treebank. We do this by collating all trees

for a given treebank across all splits that are available. We remove trees with 2 tokens or less. We then bin the trees by sentence length and by the mean edge displacement (MED) of each sentence. MED is defined as:

$$\text{MED} = \frac{1}{N-1} \sum_{n \in N} s^n_{edge} \tag{5.7}$$

where n is a given node in a given tree, N is the total number of nodes in the tree, and s^n_{edge} is the edge displacement of a given node as defined in Equation 5.6. Note the denominator is $N-1$ as the root node is not included.

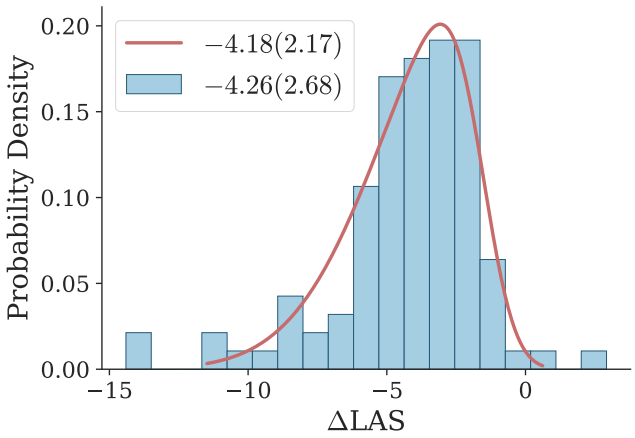


Figure 5.19: Distribution of ΔLAS (the LAS obtained from split where EDV is minimised minus the LAS obtained for the split where EDV is maximised)) for UDPipe 1.2 models trained using UD v2.7 (103 treebanks). Shown is a fit used to obtain a more conservative measure of the variance between splits with $\chi^2 = 0.40$ and p-value=0.820 (note H_0 is that the data comes from the distribution described by the fit).

We initialise the process by selecting a sentence length at random and also an MED value that exists for that sentence length bin. We then sample 3 more sentences with the closest sentence length and closest MED value available. This gives us 4 sentences with the same (or similar) sentence length and the same (or similar) MED. These are added to the training trees. We then either sample a sentence to match the MED value (when trying to keep EDV low) or sample a sentence with the furthest MED value available for the current sentence length bin (or closest if no sentences are left in a given bin) in order to maximise EDV. We repeat this process with the subsequent MED values chosen for the training instances to match the overall MED of the current training data. We do this until we have split the whole data into 80% training data and 20% test data. We then split the training data so as to obtain development data such that the overall split is 60|20|20 for training, dev, and test data, respectively. Note, we use MED and sample by tree as a more direct use of EDV would require the creation of many samples and hoping that one serendipitously maximises/minimises EDV. One could also potentially use an evolutionary algorithm to find splits which maximise (or miminise) EDV, but it would likely be computationally expensive.

We then train models using UDPipe 1.2 for the minimised EDV split and the maximised EDV split. We do this for all treebanks that have a training set of 100 sentences or more in the original split.

Sampling results

Figure 5.19 shows the distributions of ΔLAS ($\text{LAS}_{min} - \text{LAS}_{max}$) for each treebank. We fit the distribution with a skewed Gaussian function to better evaluate variance seen in this process (a more conservative one at least). When evaluating the mean of the data itself we see a mean ΔLAS of −4.26 (2.17) whereas the fit is slightly lower and with a higher

Variable	Target	Covar.	ρ	CI95%	ρ^2	Adjusted ρ^2	p-value	power
Train Tokens			0.104	[-0.09, 0.29]	0.011	-0.009	0.295	0.182
$\langle L_{\text{test}} \rangle$			0.507	[0.35, 0.64]	0.258	0.243	<0.001	1.000
ΔTokens	ΔLAS	—	0.067	[-0.13, 0.26]	0.004	-0.015	0.502	0.103
$\Delta\langle L_{\text{test}} \rangle$			-0.037	[-0.23, 0.16]	0.001	-0.019	0.713	0.065
ΔEDV			-0.478	[-0.61, -0.31]	0.228	0.213	<0.001	0.999
$\langle L_{\text{test}} \rangle$	ΔEDV		-0.847	[-0.89, -0.78]	0.717	0.711	<0.001	1.000
ΔEDV	ΔLAS	$\langle L_{\text{test}} \rangle$	-0.271	[-0.44, -0.08]	0.074	0.055	0.006	0.798

Table 5.8: Correlations between variables of interest with respect to ΔLAS using UD v2.7 (103 treebanks) and UDPipe 1.2. Shown are the coefficients (ρ), the 95% confidence intervals (CI95%), ρ^2 as an indication of the proportion of explained variance, the adjusted ρ^2 (Adjusted ρ^2) as a less biased version of ρ^2, the corresponding p-values, and the achieved power of the tests (power). The mean absolute ΔTokens is 45.0 (68.7) which is a relative difference 0.097% (with respect to the split where EDV is minimised). Mean absolute $\Delta\langle L_{\text{test}} \rangle$is 0.059 (0.172) which is a relative difference of 0.25% wrt. Tokens and $\langle L_{\text{test}} \rangle$ used here are the average across both splits.

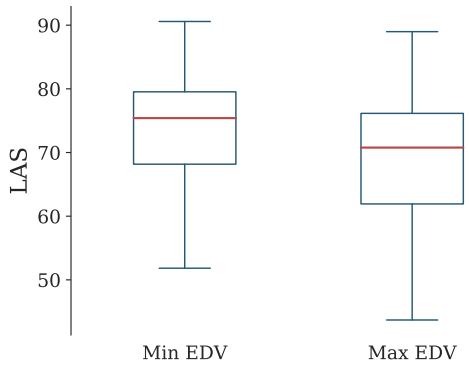


Figure 5.20: Distribution of LAS for UDPipe 1.2 models trained on splits sampled so as to miminise EDV (Min EDV) and sampling so as to maximise EDV (Max EDV) using UD v2.7 (103 treebanks). The median LAS for Min EDV is 75.40 and 70.77 for Max EDV.

standard deviation at −4.18 (2.68). This difference is considerable with typical claims of state-of-the-art performance coming down to tenths of a LAS point, so this process certainly gives a good range of performance across treebanks. Figure 5.20 shows the actual distribution of LAS values for both sets of splits. The median values of LAS are 75.40 and 70.77 for the minimum and maximum EDV splits, respectively. Note too that the spread across the first and second quartile is wider for the maximum EDV splits and with the lower tail being much smaller than that of the minimum split.

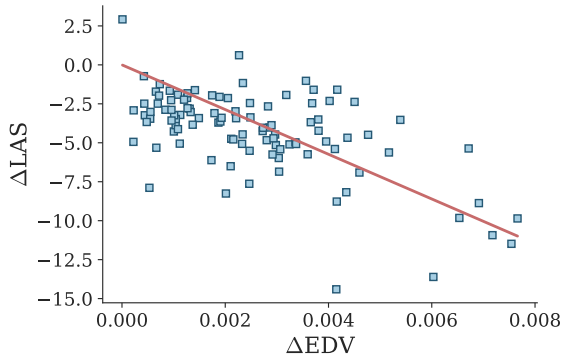


Figure 5.21: ΔLAS against ΔEDV where both are the value associated with the split where EDV has been minimised minus that of the split where EDV has been maximised. For UDPipe 1.2 models using UD v2.7 (103 treebanks).

We also evaluate whether this difference in performance can be attributed to the differences in EDV between the split. Figure 5.21 shows ΔLAS against ΔEDV ($\text{EDV}_{min}-\text{EDV}_{max}$). A strong negative linear relationship is observed as expected. To validate this observation, we once again turn to correlation coefficients. These are reported in Table 5.8. We look at the variables deemed most pertinent to evaluate EDV from the preceding analysis in Section 5.4.2. In this context, the number of training tokens (here we take the mean across the splits as an approximation) is not associated with the difference in performance across splits, which would only likely be the case if this had a major role in constraining the maximisation of EDV. Similarly the difference between the number of training tokens is not correlated to ΔLAS (the difference between splits is not large at a mean relative difference of 0.097%).

However, $\langle L_{\text{test}} \rangle$ (defined as the mean across splits) is strongly correlated to ΔLAS (ρ=0.507, p-value<0.001) and even more so to ΔEDV (ρ=0.847, p-value<0.001). This is likely due to the dependence on sentence length to vary EDV (see Figure 5.11 and Figure 5.35 in Appendix 5.A.2). However, the difference between $\langle L_{\text{test}} \rangle$ for each split is not correlated to ΔLAS, meaning that the difference observed is not merely due to the sampling procedure being forced to sample sentences of different length so as to maximise EDV. This is further attested to by the small mean relative difference between the splits of 0.25%.

ΔEDV is also strongly correlated to ΔLAS at −0.478 (p-value<0.001) which fits with the trend observed in Figure 5.21. We also report the partial coefficient of ΔEDV with respect to ΔLAS with $\langle L_{\text{test}} \rangle$ as a covariant. This results in a coefficient of −0.271 (p-value=0.006) which while being much less than the standard coefficient, still accounts for a decent amount of the variance observed (5.5% based on the adjusted ρ^2). However, this doesn't necessarily entail that the difference is not due to maximising EDV, i.e. sampling splits so as to maintain a similar sentence length distribution across splits but sampling randomly for each sentence length bin is likely to result in easier splits.

5.4.4 Summary

We have offered an analysis which has shown a clear correlation between the differences in the edge displacement distributions of training and test data in UD treebanks (as measured by the Vaserstein distance) and parsing performance (as measured by the labelled attachment

score) by using a number of methods to falsify this hypothesis. We attempted to remove signals associated with covariants which were also correlated with LAS, but still observed a linear relationship between EDV and a normalised LAS. We use statistical methods to first evaluate the partial correlations of EDV and LAS when accounting for covariants and still observed meaningful coefficients. We also used multilinear regression to evaluate whether EDV adds any predictive power to models using these same covariants and measured small but meaningful contributions from EDV. In addition, we evaluated this linear model by training new parsers with one of the systems under investigation here on treebanks in the most recent release of UD which did not already have a model and obtained predictions that were not outlandish, especially for higher performing treebanks. Further, we evaluated the partial coefficients for EDV when undertaking a sentence-length binning analysis and observed even stronger coefficients for sentences of moderate length with a clear monotonic relationship between the magnitude of the correlation of EDV to LAS with respect to sentence length.

Finally, we have shown the potential for using EDV to create splits to evaluate a best-case and a worst-case (based on the available data) scenario that is likely to be more indicative of real-world usage of parsers where out-of-domain, unseen syntactic structures likely occur in the outer regions of the distributions seen in narrow training data sets. We envisage this analysis also being useful for other practices in NLP. For example it could be used for evaluating difficulty of a given instance for curriculum learning for training parsers or for other NLP tasks, i.e. batches measured for EDV based on the overall distribution in the training data.

5.5 Conclusion

In Section 5.3, we have introduced the concept of distribution over dependency displacements that transition-based algorithms are inherently biased towards generating based on the available transitions. We then showed that similarity between these distributions and the distributions found in the test data of treebanks was related to parsing performance. In order to do so, we had to undertake a sentence length binning analysis as the correlation was not observable across the full data sample.

In Section 5.4, we took this concept and applied it to the distributions seen in training and test data and named it EDV. We found a fairly strong signal in the data between EDV and LAS, even when accounting for covariables. We also showed that the signal was stronger when undertaking a sentence length binning analysis, but that a correlation was still quite clear in the full data sample. We also showed that we could drive a sampling procedure so as to obtain adversarial and complementary splits, so as to obtain empirical limits of the performance of parsers so as to have a more robust measure of the quality of parsers.

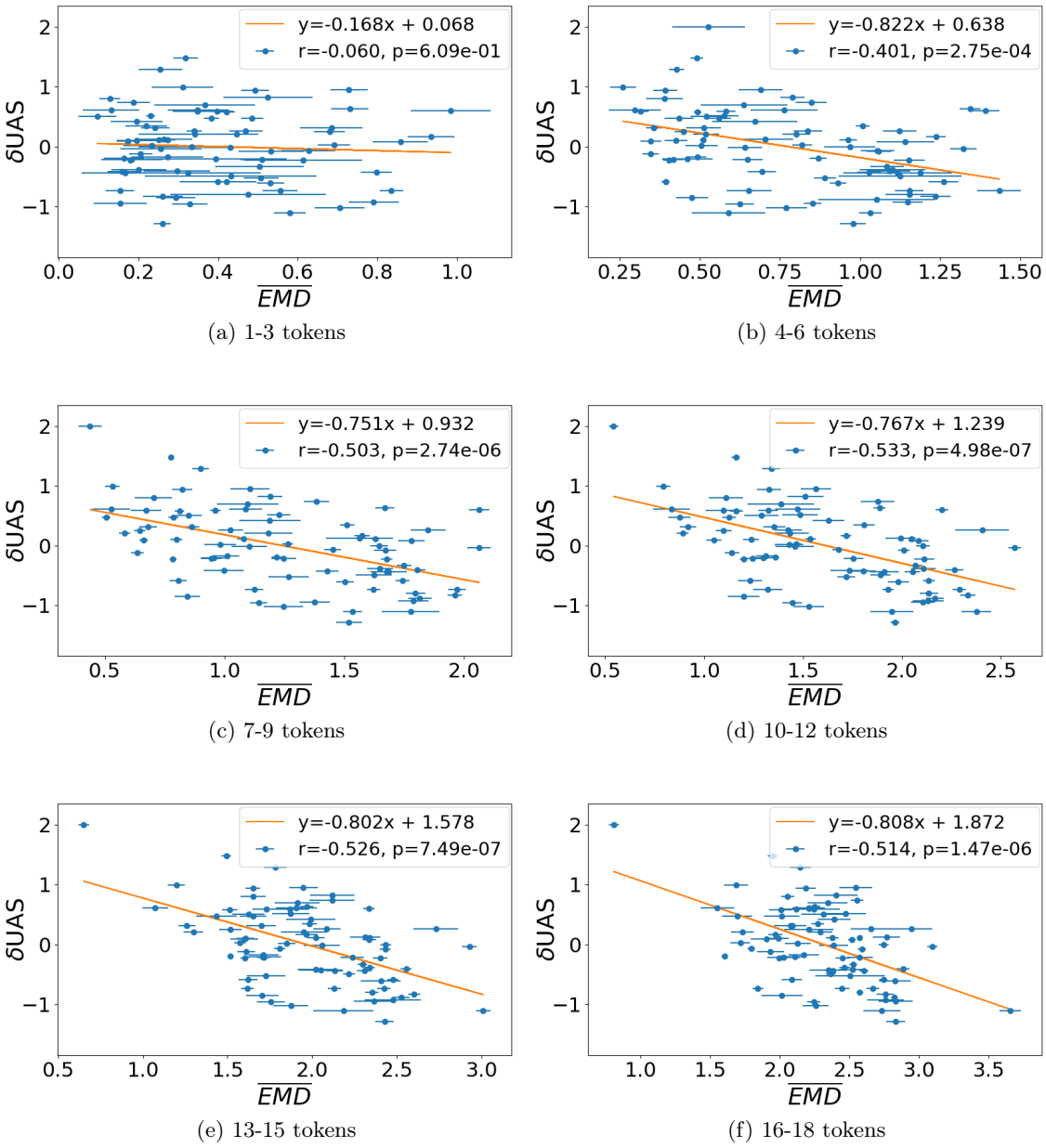
Beyond the specific interest of the analysis discussed in this chapter, we would like to highlight two general findings that could be of interest in NLP in a wider sense. First is the need to undertake robust statistical analysis, so as to avoid spurious results which often means taking into consideration potential covariants. Second, and perhaps more interestingly, we have offered two *data points* that corroborate the warnings offered by Ferrer-i-Cancho and Liu (2014), namely that sequence length can and often does influence linguistic phenomena that might be of interest to computational linguists.

5.A Appendix

5.A.1 Inherent bias appendix

We show in this section additional plots and visualisations related to the analysis of Section 5.3.

δUAS for projective algorithms



(a) 1-3 tokens

(b) 4-6 tokens

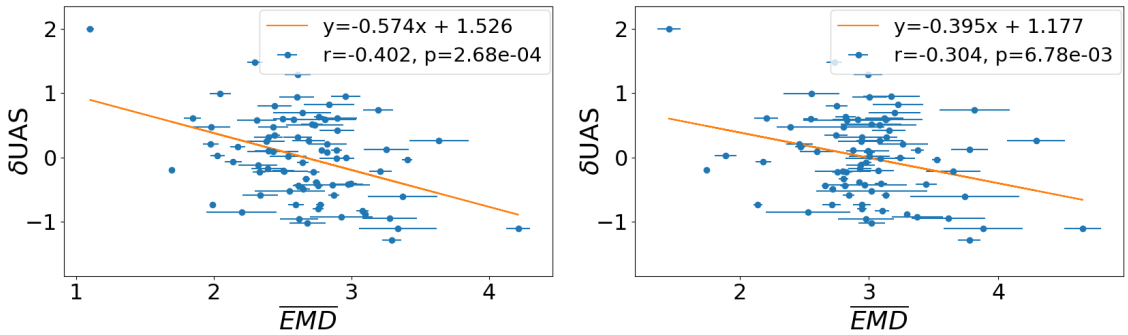
(c) 7-9 tokens

(d) 10-12 tokens

(e) 13-15 tokens

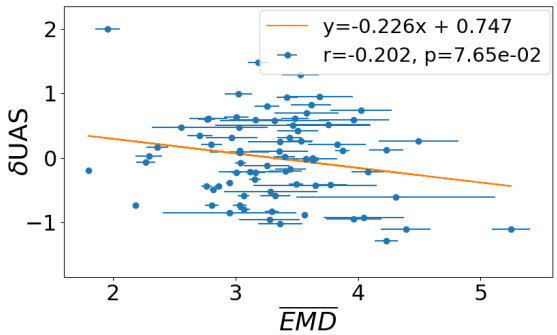
(f) 16-18 tokens

Figure 5.22: δUAS (as defined in Equation 5.3) for each algorithm against the corresponding average EMD (k=10) for projective algorithms (Arc Standard, Arc Eager, and Covington) for each sentence-length bin with 1-3 (a), 4-6 (b), 7-9 (c), 10-12 (d), 13-15 (e), and 16-18 (f) tokens with their respective Pearson coefficients (r) and the corresponding p-values (p).

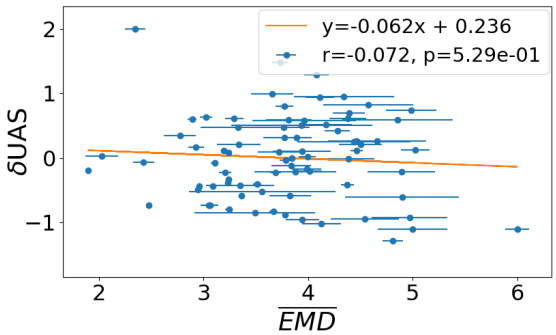


(a) 19-21 tokens

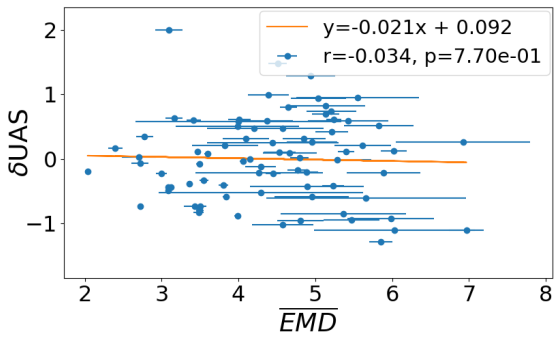
(b) 22-24 tokens



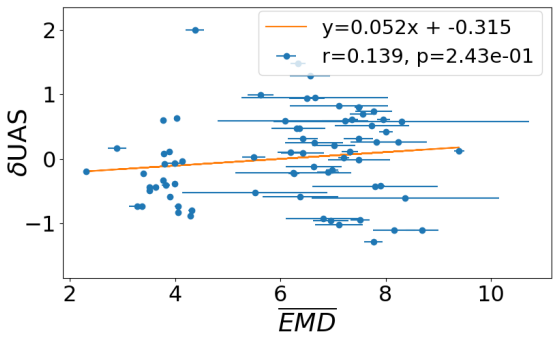
(c) 25-27 tokens



(d) 28-33 tokens



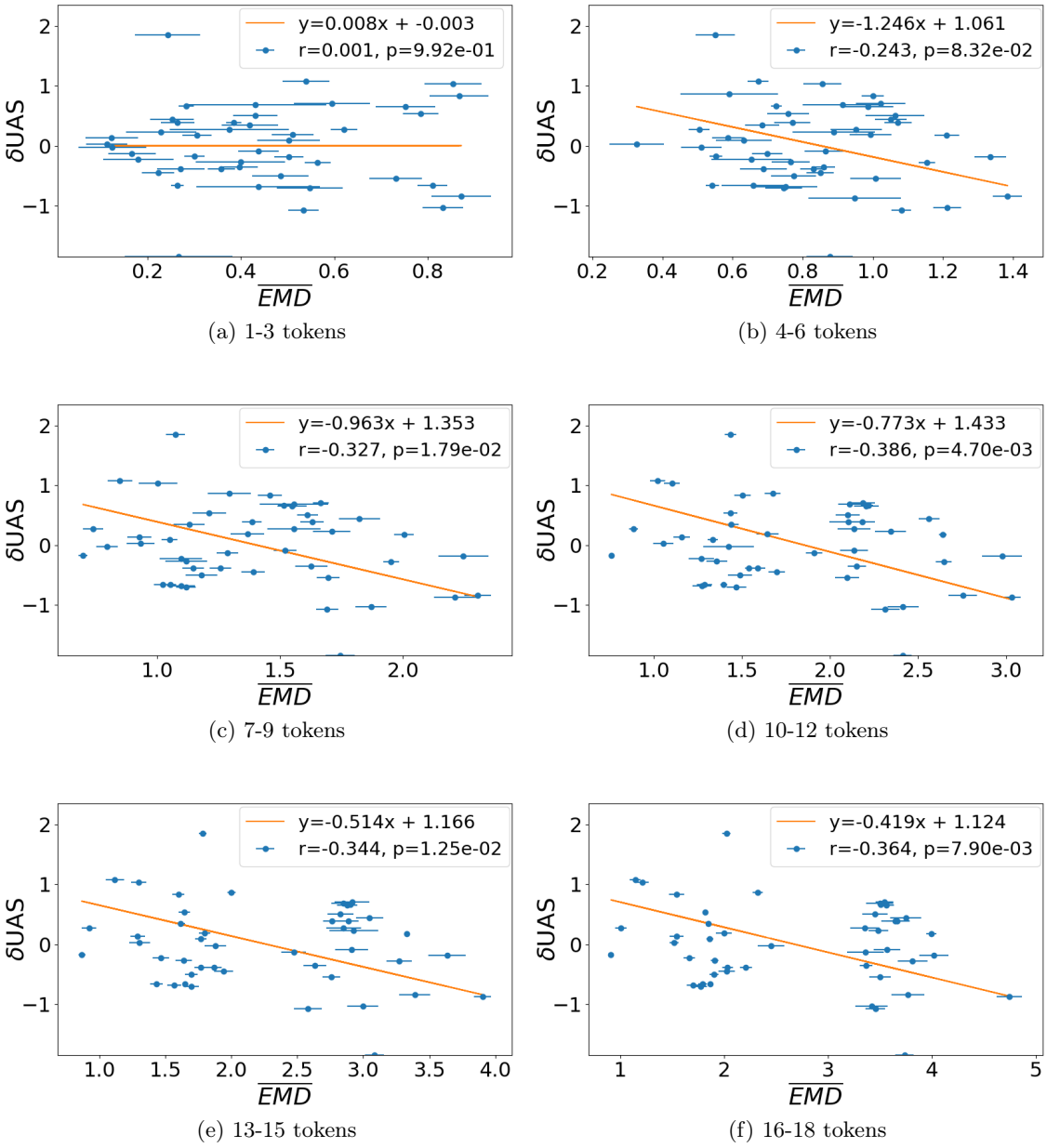
(e) 34-39 tokens



(f) 40-99 tokens

Figure 5.23: δUAS (as defined in Equation 5.3) for each algorithm against the corresponding average EMD (k=10) for projective algorithms (Arc Standard, Arc Eager, and Covington) for each sentence-length bin with 19-21 (a), 22-24 (b), 25-27 (c), 28-33 (d), 34-39 (e), and 40-99 (f) tokens with their respective Pearson coefficients (r) and the corresponding p-values (p).

δUAS for non-projective algorithms



(a) 1-3 tokens

(b) 4-6 tokens

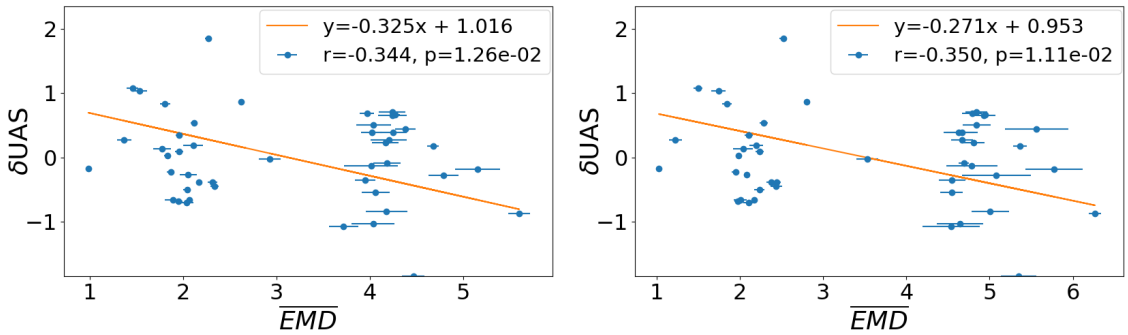
(c) 7-9 tokens

(d) 10-12 tokens

(e) 13-15 tokens

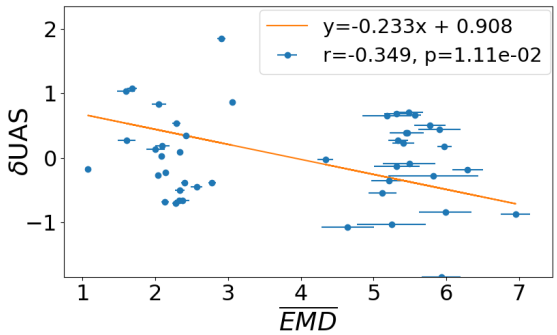
(f) 16-18 tokens

Figure 5.24: δUAS (as defined in Equation 5.3) for each algorithm against the corresponding average EMD (k=10) for non-projective algorithms (Swap Eager and non-projective Covington) for each sentence-length bin with 1-3 (a), 4-6 (b), 7-9 (c), 10-12 (d), 13-15 (e), and 16-18 (f) tokens with their respective Pearson coefficients (r) and the corresponding p-values (p). For longer sentences (13 - 18 tokens) it appears that the inherent distributions of algorithms are either quite similar or quite different with little in the middle.

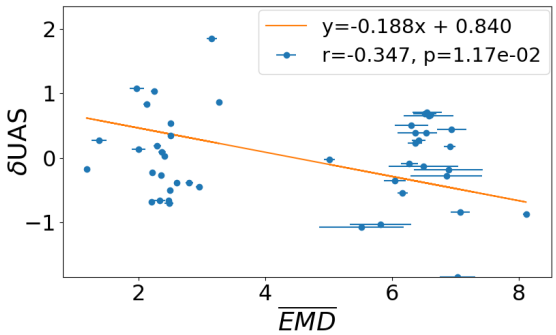


(a) 19-21 tokens

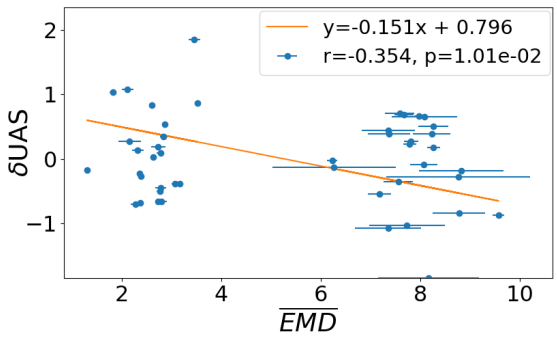
(b) 22-24 tokens



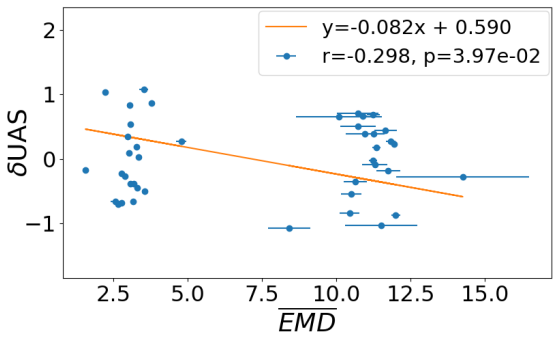
(c) 25-27 tokens



(d) 28-33 tokens



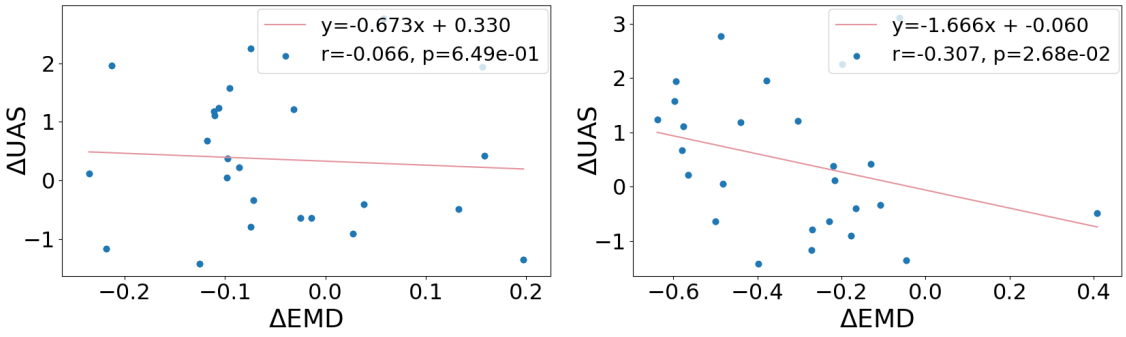
(e) 34-39 tokens



(f) 40-99 tokens

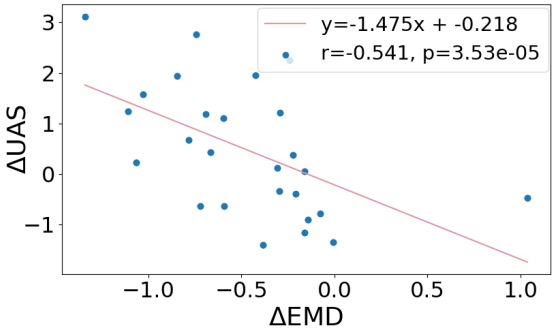
Figure 5.25: δUAS (as defined in Equation 5.3) for each algorithm against the corresponding average EMD (k=10) for non-projective algorithms (Swap Eager and non-projective Covington) for each sentence-length bin with 19-21 (a), 22-24 (b), 25-27 (c), 28-33 (d), 34-39 (e), and 40-99 (f) tokens with their respective Pearson coefficients (r) and the corresponding p-values (p). The phenomena observed in the moderate-lengthed sentence in Figure 5.24 is more apparent for these sentence-length bins so that the inherent distributions of algorithms are either quite similar or very different with few EMD values in the middle.

Arc Eager compared with Arc Standard

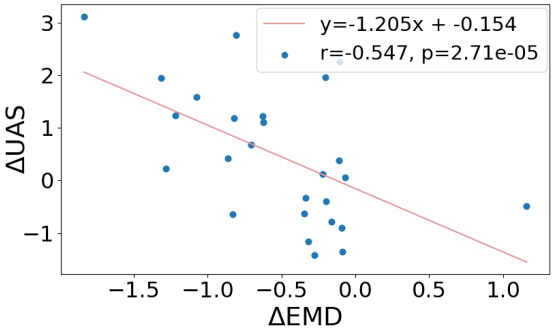


(a) 1-3 tokens

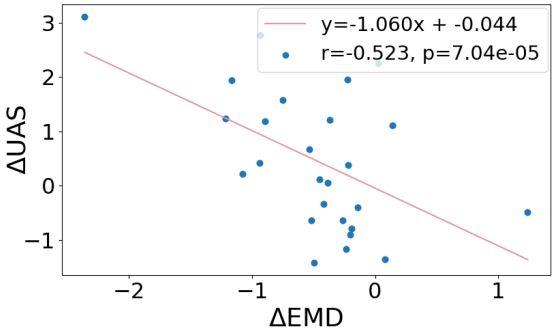
(b) 4-6 tokens



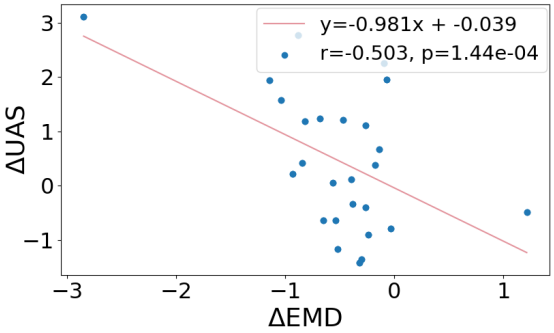
(c) 7-9 tokens



(d) 10-12 tokens

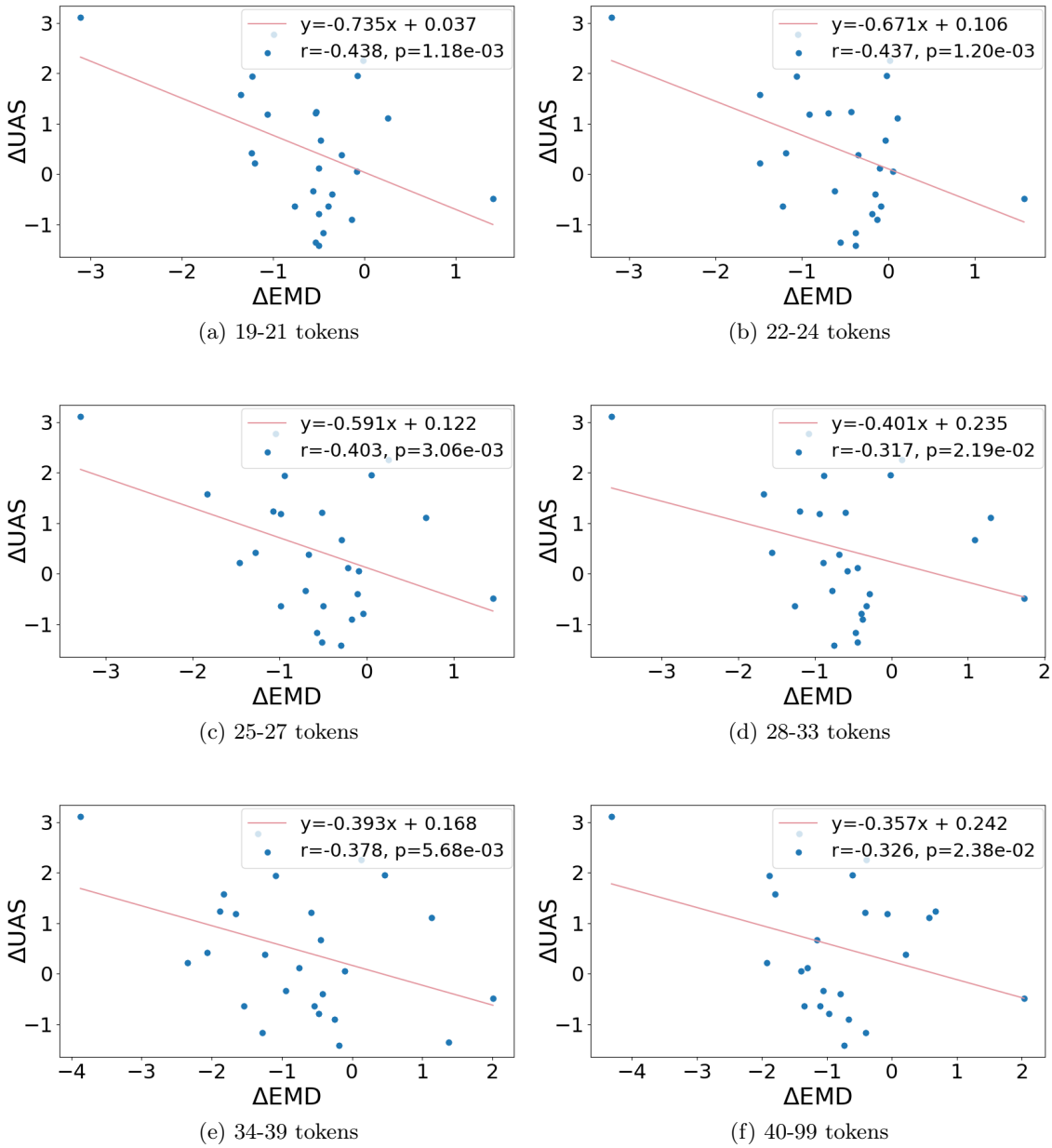


(e) 13-15 tokens



(f) 16-18 tokens

Figure 5.26: ΔUAS as defined in Equation 5.4 against ΔEMD as defined in Equation 5.5 comparing Arc Eager and Arc Standard for each sentence-length bin with 1-3 (a), 4-6 (b), 7-9 (c), 10-12 (d), 13-15 (e), and 16-18 (f) tokens with their respective Pearson coefficients (r) and the corresponding p-values (p).



(a) 19-21 tokens

(b) 22-24 tokens

(c) 25-27 tokens

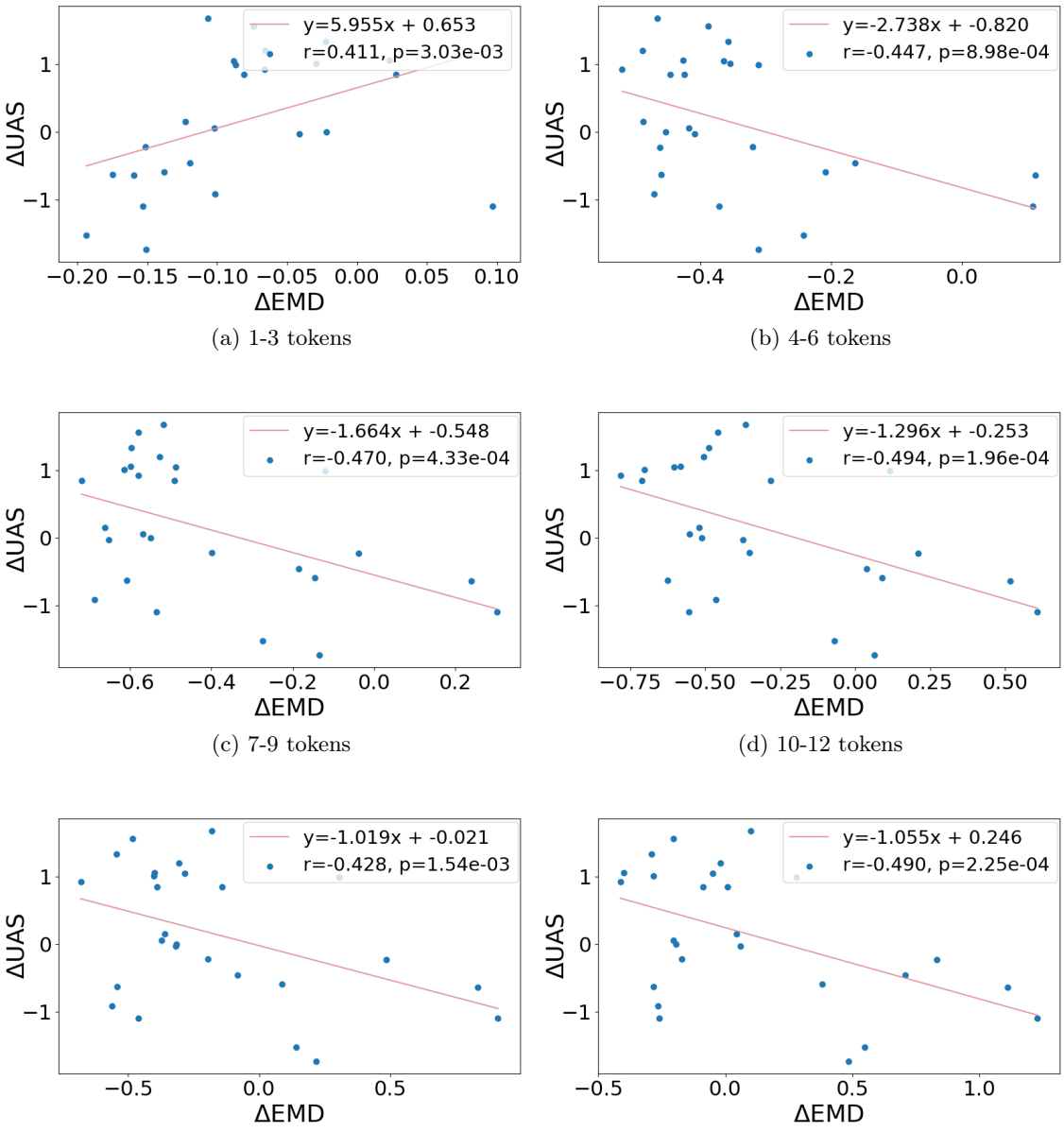
(d) 28-33 tokens

(e) 34-39 tokens

(f) 40-99 tokens

Figure 5.27: ΔUAS as defined in Equation 5.4 against ΔEMD as defined in Equation 5.5 comparing Arc Eager and Arc Standard for each sentence-length bin with 19-21 (a), 22-24 (b), 25-27 (c), 28-33 (d), 34-39 (e), and 40-99 (f) tokens with their respective Pearson coefficients (r) and the corresponding p-values (p).

Arc Standard compared with Covington (projective)



(a) 1-3 tokens

(b) 4-6 tokens

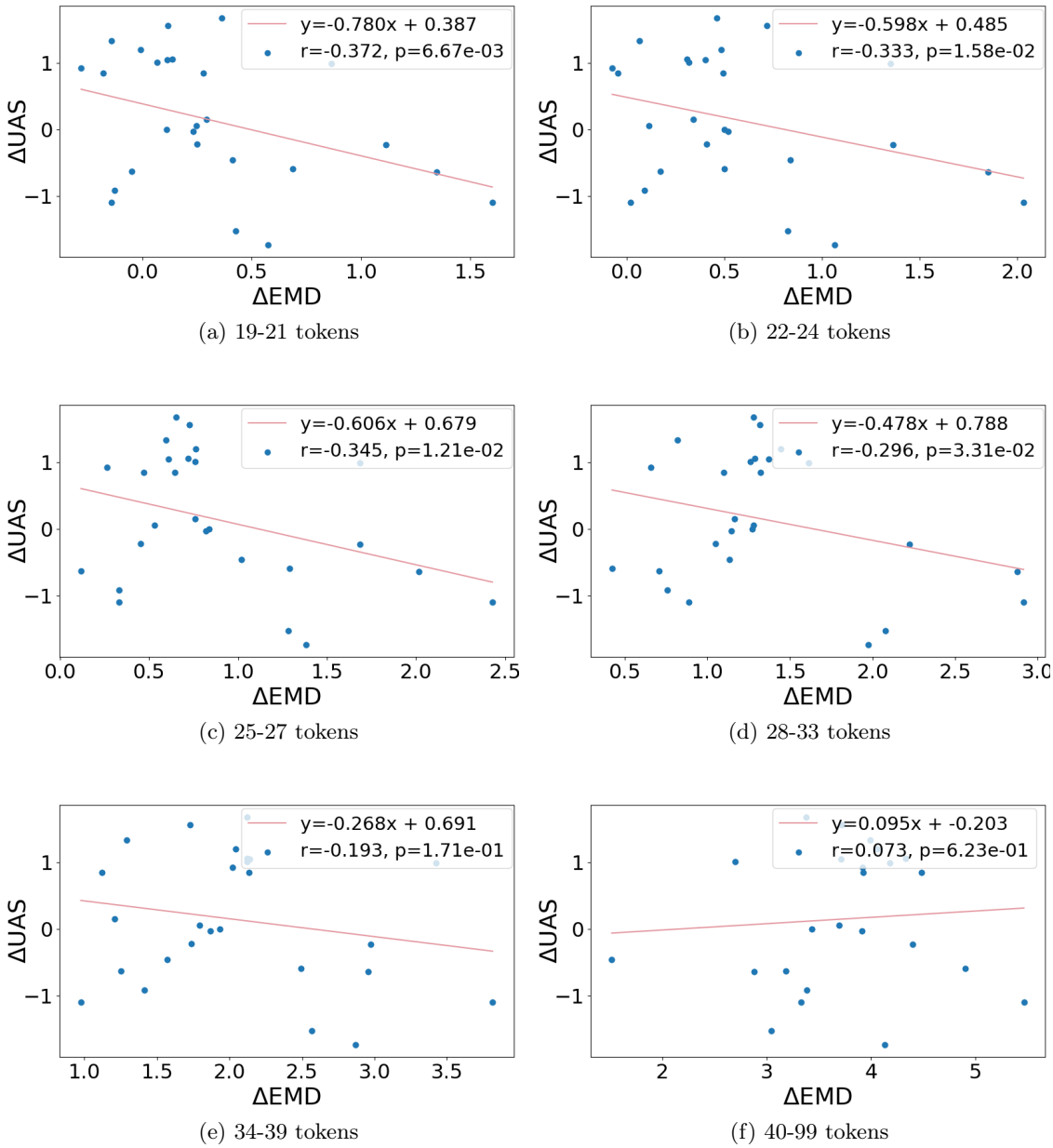
(c) 7-9 tokens

(d) 10-12 tokens

(e) 13-15 tokens

(f) 16-18 tokens

Figure 5.28: ΔUAS as defined in Equation 5.4 against ΔEMD as defined in Equation 5.5 comparing Arc Standard and Covington (projective) for each sentence-length bin with 1-3 (a), 4-6 (b), 7-9 (c), 10-12 (d), 13-15 (e), and 16-18 (f) tokens with their respective Pearson coefficients (r) and the corresponding p-values (p).



(a) 19-21 tokens

(b) 22-24 tokens

(c) 25-27 tokens

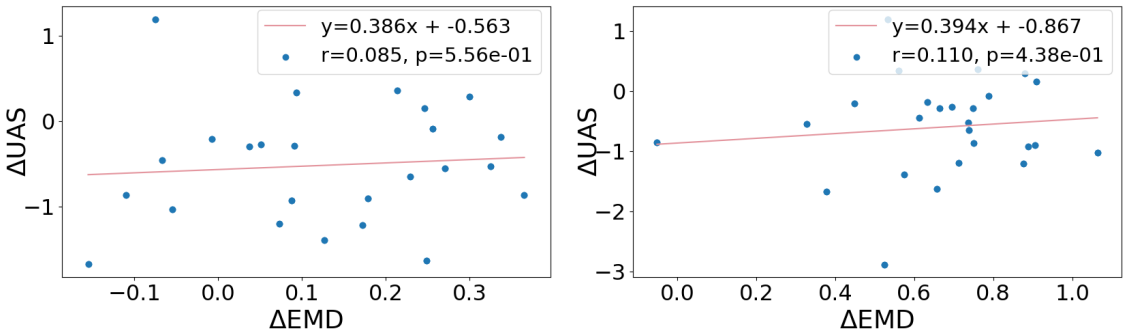
(d) 28-33 tokens

(e) 34-39 tokens

(f) 40-99 tokens

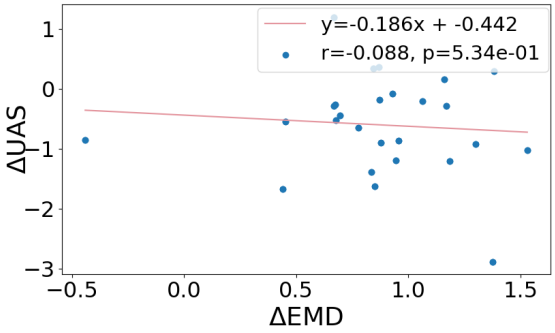
Figure 5.29: ΔUAS as defined in Equation 5.4 against ΔEMD as defined in Equation 5.5 comparing Arc Standard and Covington (projective) for each sentence-length bin with 19-21 (a), 22-24 (b), 25-27 (c), 28-33 (d), 34-39 (e), and 40-99 (f) tokens with their respective Pearson coefficients (r) and the corresponding p-values (p).

Covington (projective) compared with Arc Eager

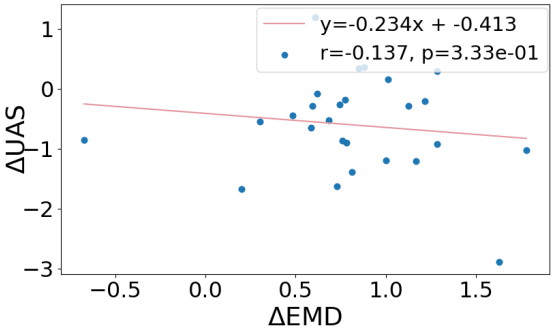


(a) 1-3 tokens

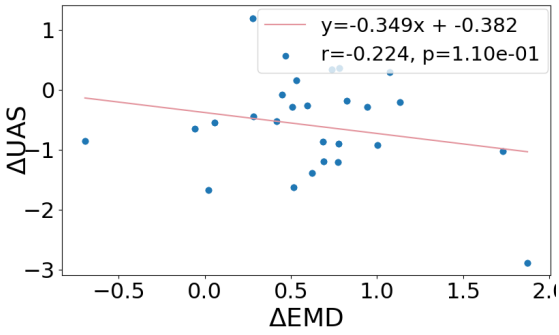
(b) 4-6 tokens



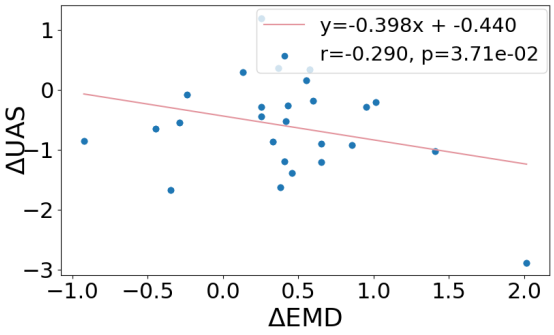
(c) 7-9 tokens



(d) 10-12 tokens

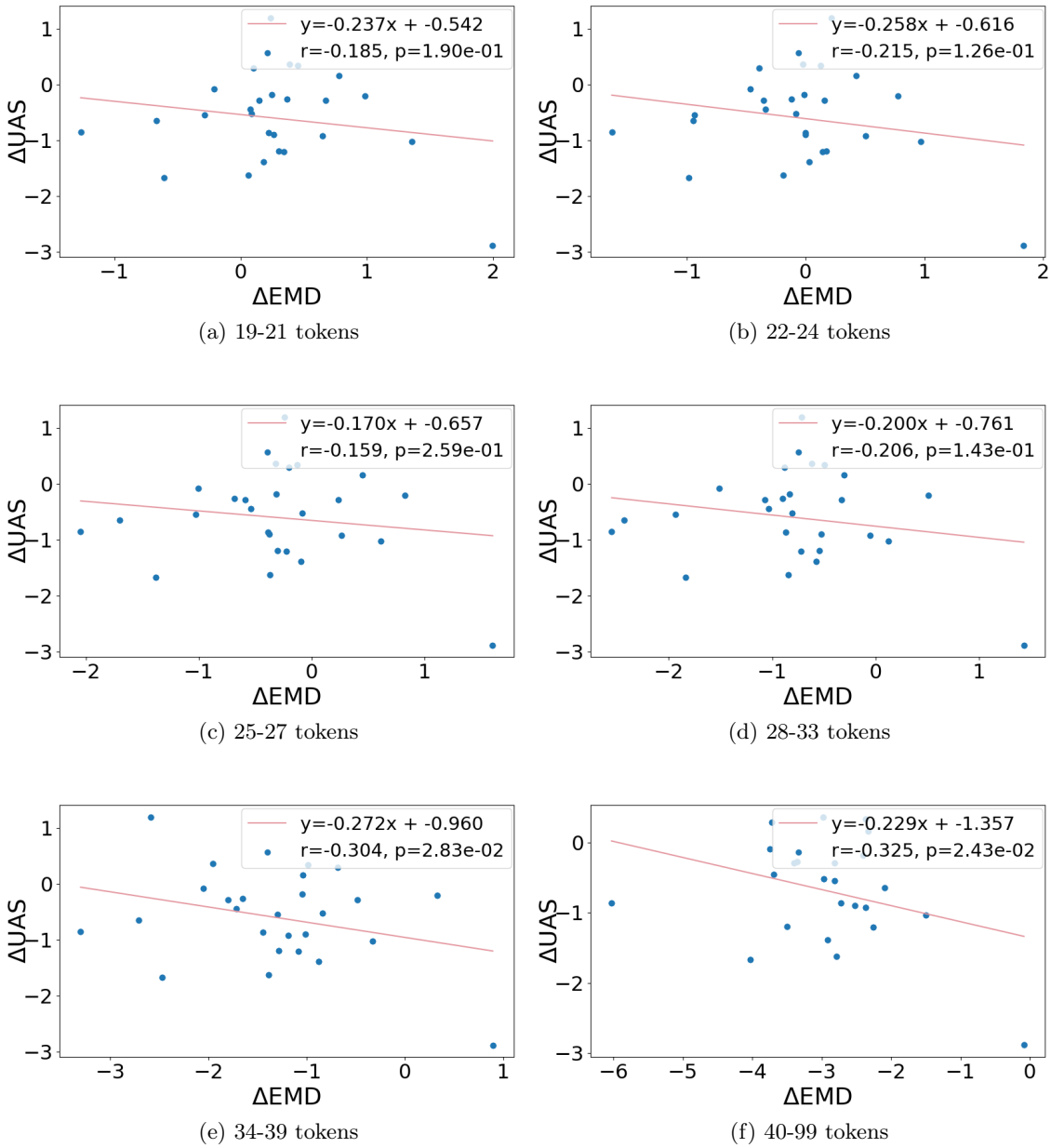


(e) 13-15 tokens



(f) 16-18 tokens

Figure 5.30: ΔUAS as defined in Equation 5.4 against ΔEMD as defined in Equation 5.5 comparing Covington (projective) and Arc Eager for each sentence-length bin with 1-3 (a), 4-6 (b), 7-9 (c), 10-12 (d), 13-15 (e), and 16-18 (f) tokens with their respective Pearson coefficients (r) and the corresponding p-values (p).



(a) 19-21 tokens

(b) 22-24 tokens

(c) 25-27 tokens

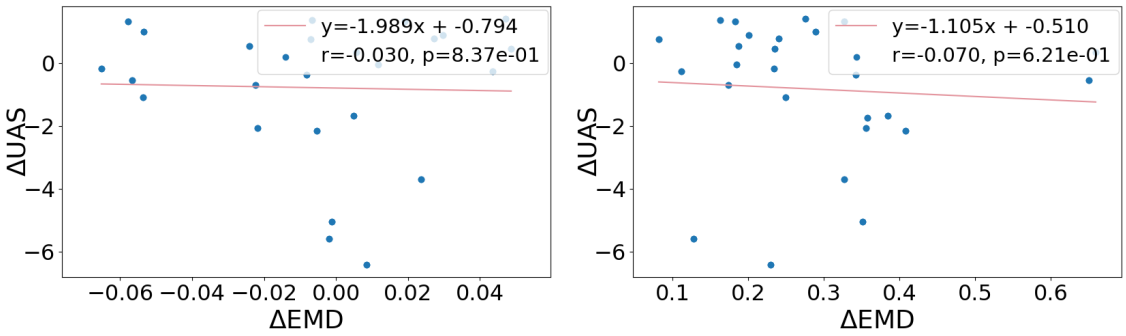
(d) 28-33 tokens

(e) 34-39 tokens

(f) 40-99 tokens

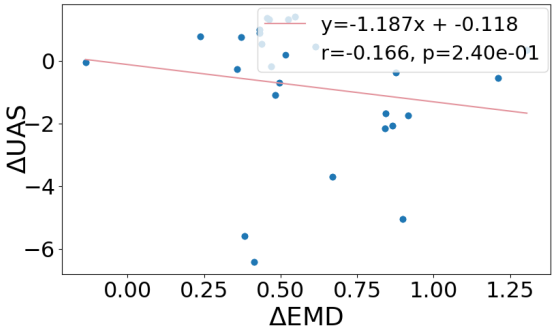
Figure 5.31: ΔUAS as defined in Equation 5.4 against ΔEMD as defined in Equation 5.5 comparing Covington (projective) and Arc Eager for each sentence-length bin with 19-21 (a), 22-24 (b), 25-27 (c), 28-33 (d), 34-39 (e), and 40-99 (f) tokens with their respective Pearson coefficients (r) and the corresponding p-values (p).

Swap Eager compared with non-projective Covington

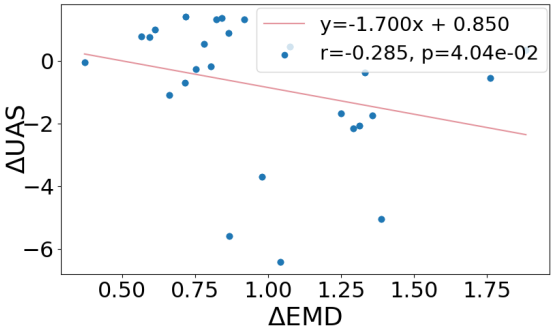


(a) 1-3 tokens

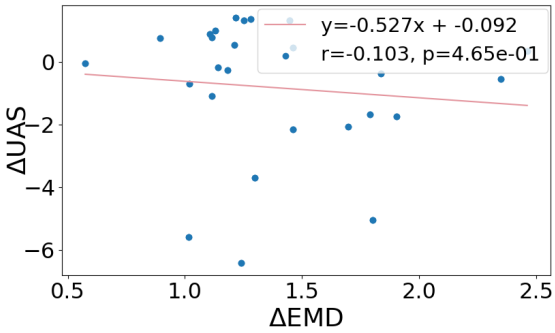
(b) 4-6 tokens



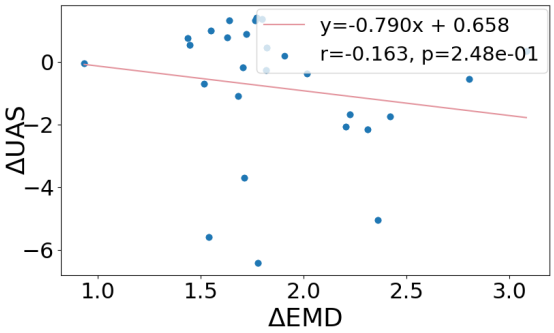
(c) 7-9 tokens



(d) 10-12 tokens



(e) 13-15 tokens



(f) 16-18 tokens

Figure 5.32: ΔUAS as defined in Equation 5.4 against ΔEMD as defined in Equation 5.5 comparing Swap Eager and non-projective Covington for each sentence-length bin with 1-3 (a), 4-6 (b), 7-9 (c), 10-12 (d), 13-15 (e), and 16-18 (f) tokens with their respective Pearson coefficients (r) and the corresponding p-values (p).

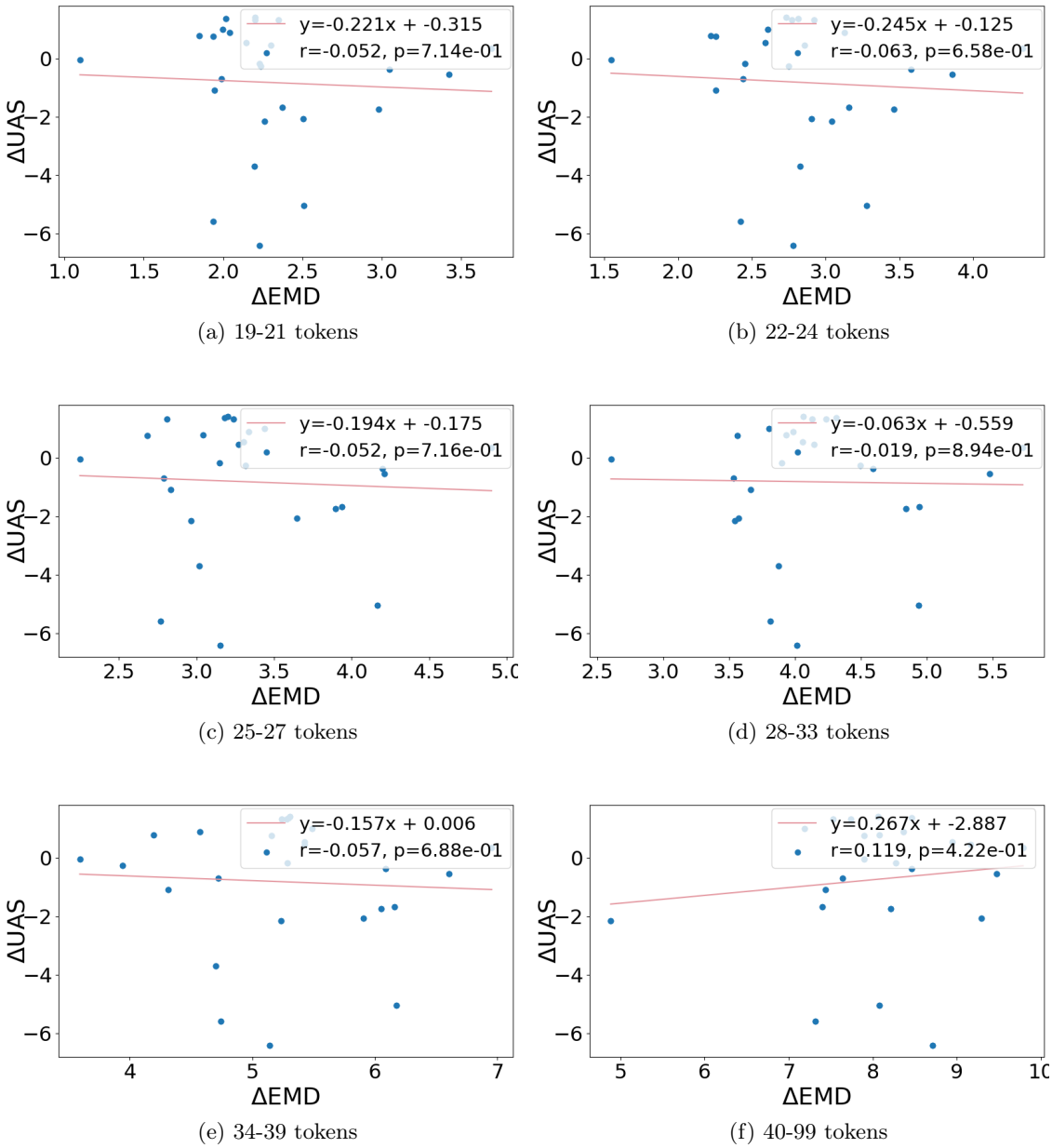


Figure 5.33:]
ΔUAS as defined in Equation 5.4 against ΔEMD as defined in Equation 5.5 comparing Swap Eager and non-projective Covington for each sentence-length bin with 19-21 (a), 22-24 (b), 25-27 (c), 28-33 (d), 34-39 (e), and 40-99 (f) tokens with their respective Pearson coefficients (r) and the corresponding p-values (p).

5.A.2 EDV appendix

This appendix is mainly for showing the corresponding data for UDPipe 1.2 (as we showed the data for UDPipe 2.0 for the most part in the main text) for Section 5.4. Almost universally the observed behaviour follows that shown in the main text. If it had been otherwise, we would have opted to show conflicting data visualisations. Figures 5.40 and 5.41 show the data used to evaluate the coefficients shown in Figure 5.18.

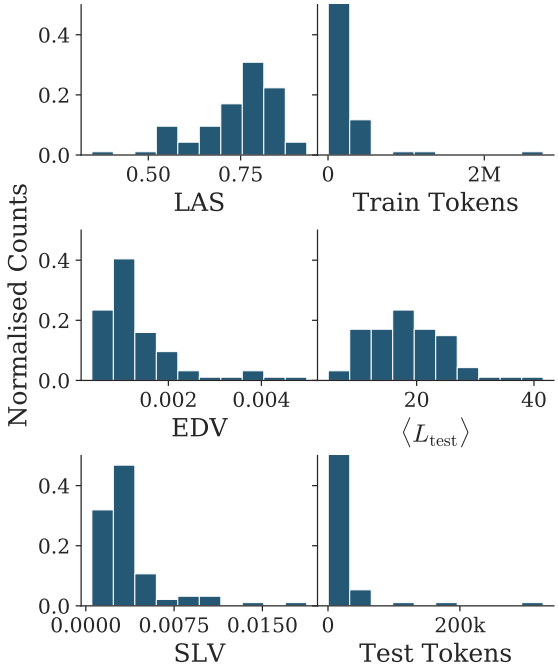


Figure 5.34: Distributions of the variables of interest in UD v2.5 (94 treebanks) in order to evaluate whether they are sampled from normal distributions.

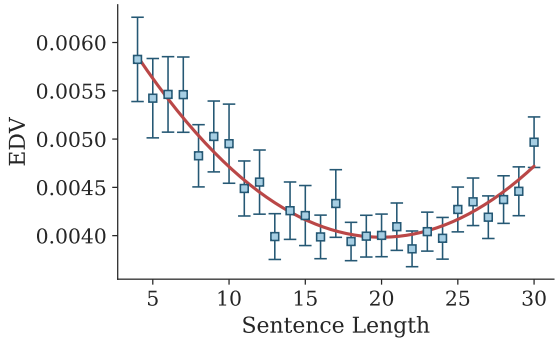


Figure 5.35: EDV between sub-samples of the training and test data binned by sentence length for UD v2.5 (105 treebanks).

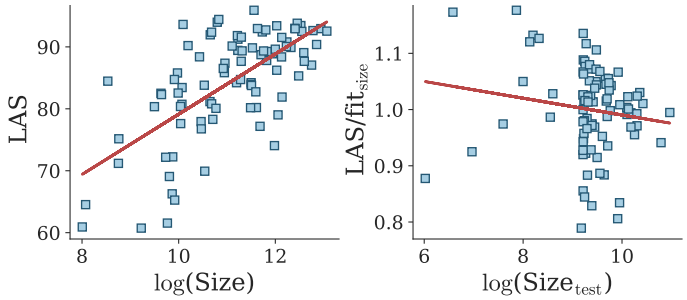


Figure 5.36: Background removal method used to evaluate whether the number of test tokens carries additional information with respect to the number of training tokens for UDPipe 2.0 and UD v2.6. Correlation between the number of test tokens and LAS is 0.309 (p-value=0.003) and that between the number of test tokens and the normalised LAS (right plot) is -0.101 (p-value=0.342).

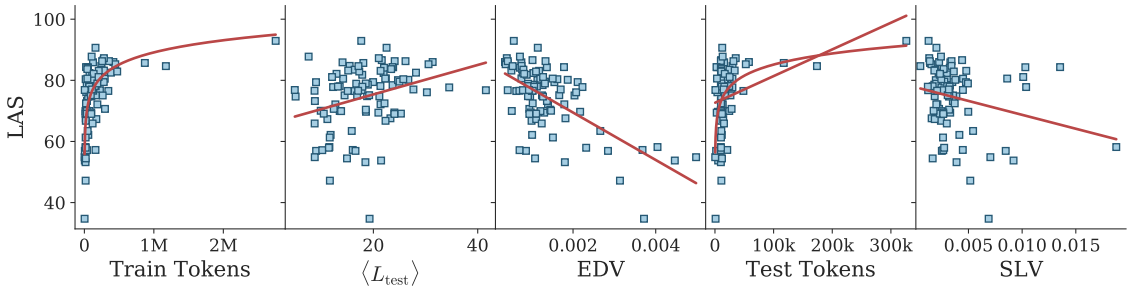


Figure 5.37: Visualisation of LAS (for UDPipe 1.2 and UD v2.5) with respect to variables of interest with fits shown in red to highlight whether the data appears correlated or not.

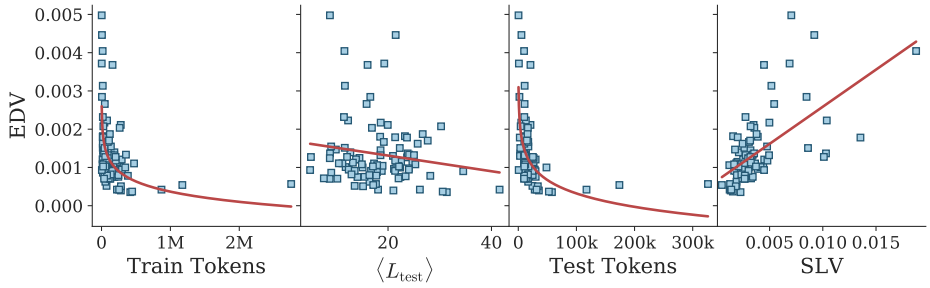


Figure 5.38: Visualisation of EDV (for UD v2.5) with respect to variables of interest with fits shown in red to highlight whether the data appears correlated or not.

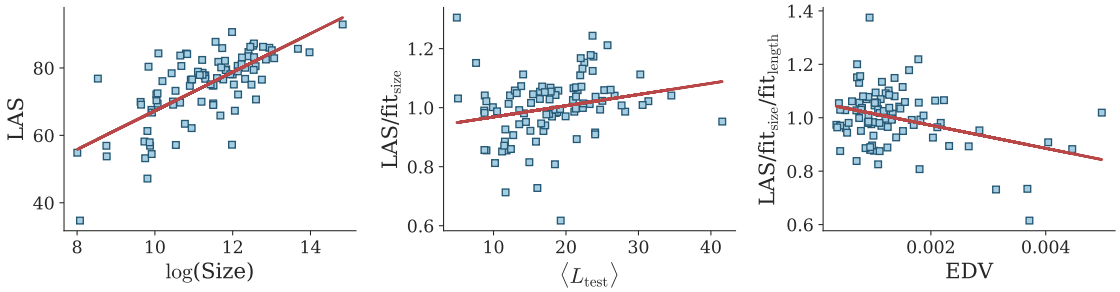


Figure 5.39: Background removal method to evaluate whether a correlation is observed between EDV and LAS (for UDPipe 1.2 and UD v2.5) after removing the variation associated with the training test size and $\langle L_{\text{test}} \rangle$. The correlation between EDV and LAS is -0.492 (p-value<0.001), the correlation between EDV and the LAS normalised by the variance associated with number of tokens in training data is -0.186 (pvalue=0.072), and the correlation for the fully normalised LAS (removing the variance associated with $\langle L_{\text{test}} \rangle$) is -0.249 (pvalue=0.015).

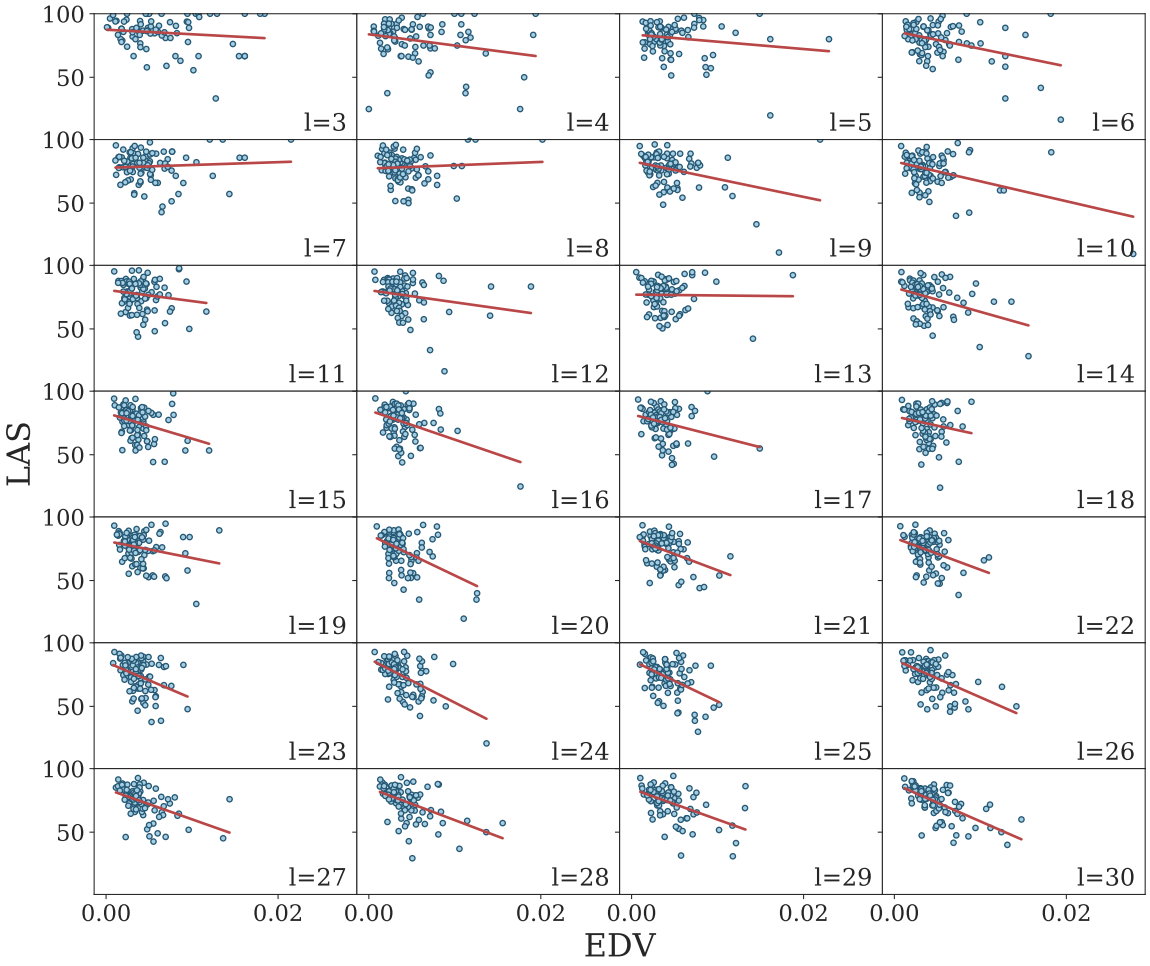


Figure 5.40: LAS versus EDV for each sentence length bin (labelled l=length) for UDPipe 1.2 used for calculating the coefficients shown in Figure 5.18.

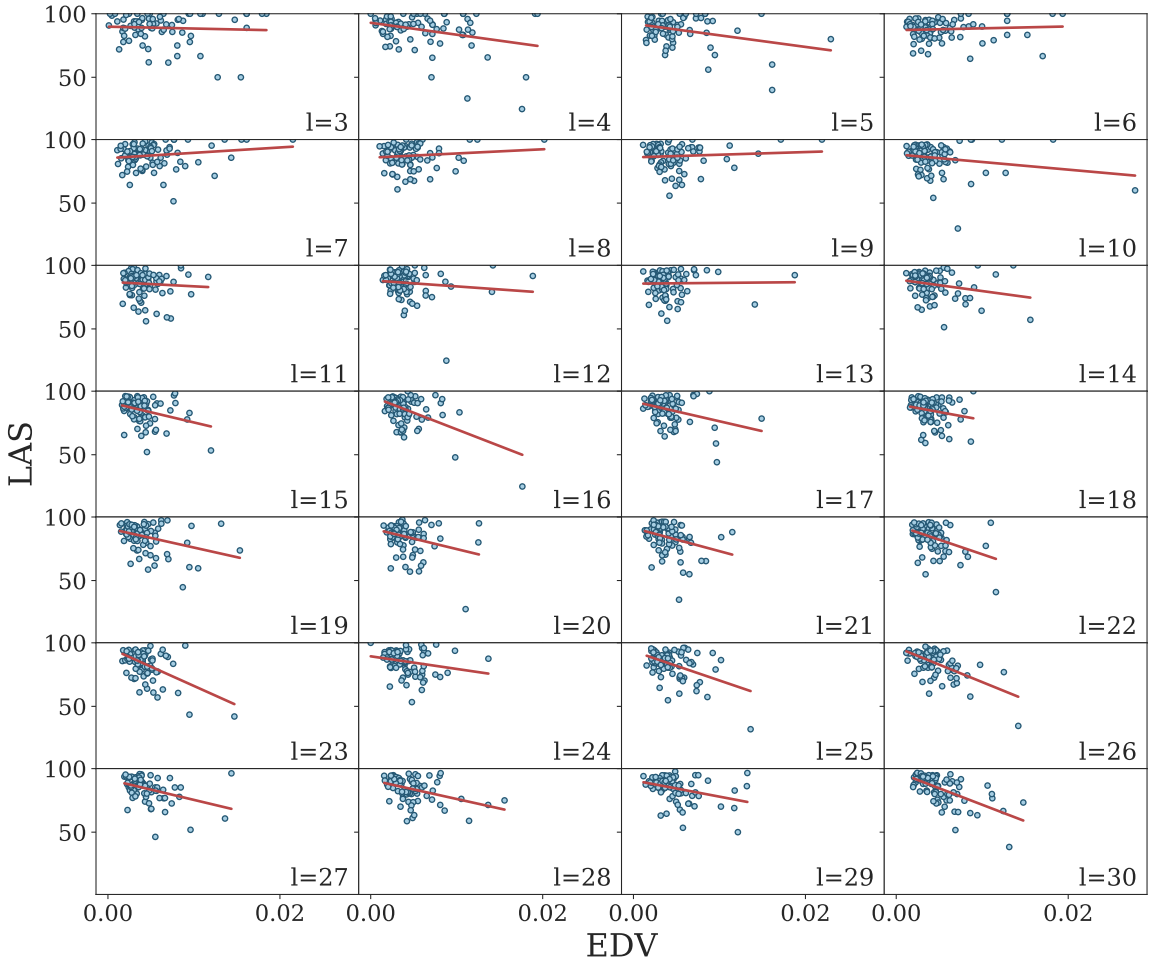


Figure 5.41: LAS versus EDV for each sentence length bin (labelled l=length) for UDPipe 2.0 used for calculating the coefficients shown in Figure 5.18.

Chapter 6

On Universal part-of-speech tags

Work in this chapter based on in part published work in Anderson and Gómez-Rodríguez (2020c) and in Anderson and Gómez-Rodríguez (2021).

This chapter focuses on the impact POS tagging accuracy has on dependency parsers when using predicted POS tags as features to neural network parsing systems. The main work focuses on controlled experiments where POS tag accuracy is considered the independent variable and parsing performance as the dependent variable. We also present work that analyses in what context taggers fail and how this relates to what neural parsers implicitly learn about POS tags. And we present work on this impact in low-resource settings.

6.1 Introduction

A part-of-speech (POS) is a category assigned to tokens which exhibit similar grammatical properties and in general similar syntactic behaviour. POS tagging is then the task of classifying tokens as a given POS tag based on the meaning and use of each token in a given context. As such, POS tagging and parsing are closely related NLP tasks and have been strongly connected in the past, with POS tagging seen as the immediate step before parsing in what is referred to as the classical NLP pipeline.

Different sets of POS tags are used depending on the framework and often are language-specific. As we are focusing on multilinguality and have used UD treebanks throughout this thesis, we use the set of tags associated with this framework. When referring to POS tags in Section 6.2 we use the term POS tags to refer to tags in a general sense and highlight when work is specifically on universal POS (UPOS) tags. However, in the sections discussing the work undertaken for this thesis, we refer to universal POS tags as POS for brevity.

We first give a brief overview of prevalent work in this area in Section 6.2. In Section 6.3, we undertake a series of controlled experiments where we purposefully alter the accuracy of taggers. Then based on the results we obtained from that work, in Section 6.4 we evaluate why gold standard POS tags increase performance much more than when using predicted tags, and in Section 6.5 we investigate the impact POS tagging accuracy has when data is limited.

6.2 Related work

Part-of-speech (POS) tags and dependency parsing have formed a long-standing union in NLP. But equally long-standing has been the question of its efficacy. Prior to the prevalence of deep learning in NLP, they were shown to be useful for syntactic disambiguation in certain

contexts (Voutilainen, 1998; Dalrymple, 2006; Alfared and Béchet, 2012). However, for neural network implementations, especially those which utilise character embeddings, POS tags have been shown to be much less useful (Ballesteros et al., 2015; de Lhoneux et al., 2017a).

Others have found that POS tags can still have a positive impact when using character representations given that the accuracy of the predicted POS tags used is sufficiently high (Dozat et al., 2017). Typically using predicted POS tags has offered a nominal increase in performance or has had no impact at all. Smith et al. (2018) undertook a thorough systematic analysis of the interplay of UPOS tags, character embeddings, and pre-trained word embeddings for multi-lingual Universal Dependency (UD) parsing and found that tags offer a marginal improvement for their transition based parser. The use of fine-grained POS tags still seems to garner noticeable improvements even for challenging multi-lingual settings (Ammar et al., 2016).

However, Zhang et al. (2020b) found that the only way to leverage POS tags (both coarse and fine-grained) for English and Chinese dependency parsing was to utilise them as an auxiliary task in a multi-task framework. They have similarly been used in neural network parsers in multi-learning frameworks where they can be leveraged without the cost of error-propagation (Zhang and Weiss, 2016; Yang et al., 2017; Li et al., 2018c; Nguyen and Verspoor, 2018). Beyond multi-learning systems, Strzyz et al. (2019b) introduced dependency parsing as sequence labelling by encoding dependencies using relative positions of UPOS tags, thus explicitly requiring them at runtime. So even if coarse POS tags, universal or otherwise, prove to be superfluous for graph- or transition-based neural parsers as direct features, there are still many uses for them in dependency parsing.

6.2.1 POS tags for low resource parsing

Low resource parsing is a long-standing problem in NLP and many techniques have been introduced to tackle it (Hwa et al., 2005; Zeman and Resnik, 2008; Ganchev et al., 2009; McDonald et al., 2011; Agić et al., 2016). For an extensive review and comparison of techniques see Vania et al. (2019). Tiedemann (2015) highlighted the unrealistic performance of low resource parsers when using gold POS tags in a simulated low resource setting. The performance difference was stark despite using fairly accurate taggers, which isn't a particularly reasonable assumption for low resource languages. Tagging performance in low resource settings is still very weak even when utilising cross-lingual techniques and other forms of weak supervision (Kann et al., 2020).

6.3 POS tag accuracy

We present an analysis on the effect UPOS accuracy has on parsing performance. Results suggest that leveraging UPOS tags as features for neural parsers requires a prohibitively high tagging accuracy and that the use of gold tags offers a non-linear increase in performance, suggesting some sort of exceptionality. We also investigate what aspects of predicted UPOS tags impact parsing accuracy the most, highlighting some potentially meaningful linguistic facets of the problem.

We follow the work of Smith et al. (2018) and evaluate the interplay of word embeddings, character embeddings, and POS tags as features for two modern parsers, one a graph-based biaffine parser, Biaffine, and the other a transition-based parser, UUParser (Dozat and Manning, 2017; Smith et al., 2018). Similar to Zhang et al. (2020b), we focus on the contribution of POS tags but evaluate UPOS tags.

6.3.1 Experimental details

We ran three experiments to measure the impact POS[1] tagging accuracy has on parsing performance when using POS tags as features. Experiment 1 considered the POS tagging accuracy as a controlled variable, set by training taggers as described below and then using the output of these taggers as features for parsers. Experiment 2 was similar, except the size of character embeddings was also changed. Experiment 3 was an extension to test the impact of taggers in an optimal setting where they achieve very high accuracies.

Data We use the same subset of UD v2.4 treebanks (Nivre et al., 2019) as Anderson and Gómez-Rodríguez (2020a): Ancient Greek Perseus, Chinese GSD, English EWT, Finnish TDT, Hebrew HTB, Russian GSD, Tamil TTB, Uyghur UDT, and Wolof WTB. We used fastText word embeddings for each language except for Ancient Greek and Wolof (Grave et al., 2018). For Ancient Greek we use embeddings from Ginter et al. (2017) and for Wolof those from Heinzerling and Strube (2018). When necessary, we reduced the dimensions to 100 using the algorithm of Raunak (2017).

6.3.2 Methodology

POS taggers We train POS taggers for each treebank separately using the sequence-labelling framework NCRF++ (Yang and Zhang, 2018). We train taggers so as to have POS taggers with varying accuracies ranging from 60 to the maximum score the network can achieve (that fits our binning procedure). The accuracy bins we used were increments of 2.5±0.3 from 60 to 80 and increments of 1±0.3 from 80 onwards. We allowed a small window around the desired accuracy for each bin to account for the fact we might never see a model with that exact accuracy. To obtain taggers with varying accuracies, we train each tagger as normal and save models when they reach a certain accuracy. We chose to vary the accuracy of the taggers in this more *natural* way so as to better represent how the taggers would likely behave if they were trained normally but never exceeded the accuracy of a given bin, so it is more likely that easier patterns are learnt first and systematic failures are more likely than if we randomly added noise.

Network details We use the default parameters for both parsers, i.e. those reported in each subsequent paper. We use v2.3 of UUParser[2] and use a PyTorch implementation of the biaffine parser.[3] The features to the networks are the word embeddings as mentioned above, character embeddings, and POS tag embeddings, with the latter two embeddings being randomly initialised. For Experiment 1, the character embedding size was 32 and varied as specified below for Experiment 2. The BiLSTM output dimension of the character embedding layer was 100 and the embedding dimension of the word and POS embeddings were also 100. These dimensions were chosen to control the contribution from each feature, but it is obviously feasible that optimising these contributions could result in different absolute results. However, keeping these static unless purposefully changing them for controlled input means we can make relative comparisons.

Experiment 1 We trained parsers for each treebank with gold tags and with predicted tags using a subset of the POS taggers with accuracy bins 60, 70, 80, 86, 91, and 93. The values were chosen such that we could cover a reasonable range and include as many treebanks as possible (e.g. only English, Hebrew, and Russian have taggers which achieve 93% accuracy). The parsers trained with predicted tags are run on inputs tagged by the same model, and those trained with gold tags are tested both on gold and a range of predicted tags. The goal

[1] From this point on we refer to universal POS tags as POS tags, rather than UPOS tags, for sake of efficiency.
[2] UUParser GitHub from Smith et al. (2018).
[3] Biaffine PyTorch GitHub based on Dozat and Manning (2017).

of this experiment was to test the sensitivity of parsers to POS tagging accuracy for different treebanks. We also trained parsers without POS tags as a baseline for comparison.

Experiment 2 We trained parsers for each treebank with gold tags and with predicted tags using a subset of the POS taggers with accuracy bins 80, 86, and the maximum accuracy for each treebank which was on average 91(3). Each parser is run on inputs tagged by the same model. We used varying character embedding sizes of 32, 100, 180, 325, and 500. We also train parsers with these varying character embedding sizes with no POS tags as a baseline.

Experiment 3 We trained parsers with and without predicted POS tags for treebanks for which we obtained high-scoring POS taggers with a mean accuracy of 96(2) to evaluate the trend observed in Experiment 1. We use the settings from Experiment 1. The treebanks used were Catalan AnCora, Japanese GSD, Latin ITTB, and Polish PDB.

6.3.3 Results and analysis

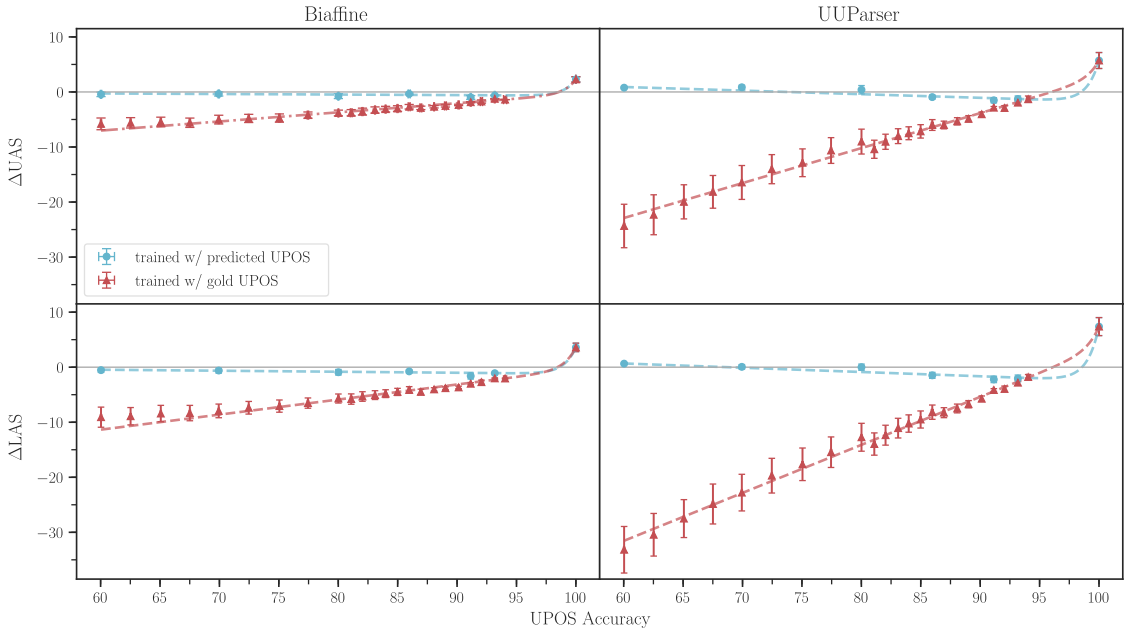


Figure 6.1: Average Δ attachment scores across all treebanks over the relative baseline parsers trained without POS tags, plotted with respect to POS tag accuracy for parsers trained with predicted (blue, circles) and gold (red, triangles) tags.

Experiment 1 The results of Experiment 1 are shown in Figure 6.1, where the average difference in attachment scores between the baseline parsers (without POS tags) and those with differing POS tag accuracies are shown. We show the differences in attachment scores rather than the absolute values, as averaging over treebanks obscures differences.

There is an unsurprising relation between parsing score and tagging performance when training with gold POS tags. What is less expected is how little of an impact is observed when using predicted tags during training, with an almost consistent performance with respect to POS tag accuracy.

The gold training trend for the graph-based parser suggests that it is less sensitive to POS tag accuracy than the transition-based parser. This is likely due to the transition-based parser being able to leverage POS tags more, so that it will see more of an impact when tagging accuracy is low. This is somewhat corroborated by the larger positive difference over the baseline when using gold tags at prediction time for UUParser compared to the increase seen for Biaffine.

Another notable phenomenon is that the results for parsing texts annotated with gold POS (rightmost point in each plot) outperform what could be expected from extrapolating the general trends. This raises the question as to whether this is due to smooth nonlinear accuracy increases in the rightmost part of the curves (where we couldn't obtain taggers) or to a sudden jump at the very end in a hockey-stick shape, indicating an exceptionality of gold POS tags and inadequacy of even very accurate but imperfect POS tags (which is relevant under the assumption that tagging accuracy can be pushed further with future model and/or training data improvements). Answering this question was the motivation for Experiment 3.

Almost exclusively, using predicted POS tags does not outperform the parser trained without any POS tags. Curiously, the only parsers that are marginally better are those trained with predicted POS tags from the least accurate POS taggers.

Figures 6.14–6.17 in Appendix 6.A.1 show these results for each treebank separately, and almost all treebanks follow the general trend seen for both parsers. The only exception is Tamil TTB for UUParser, which benefits from POS tags both when training with gold and predicted tags. Tamil TTB is the smallest treebank, and it has the additional difficulty for parsing and tagging of being an agglutinative language, so possibly this combination of factors lends itself well to leveraging POS tags even in less than optimal circumstances. Tamil is also the lowest performing language with respect to POS tagging and parsing accuracy, but compared to Uyghur and Ancient Greek (the next two lowest performing languages) it outperforms both when using gold tags. In fact, Tamil has the biggest difference when using gold tags over the baseline than any other language, suggesting that they might be particularly useful when there is a heightened probability of ambiguity coupled with a dearth of data.

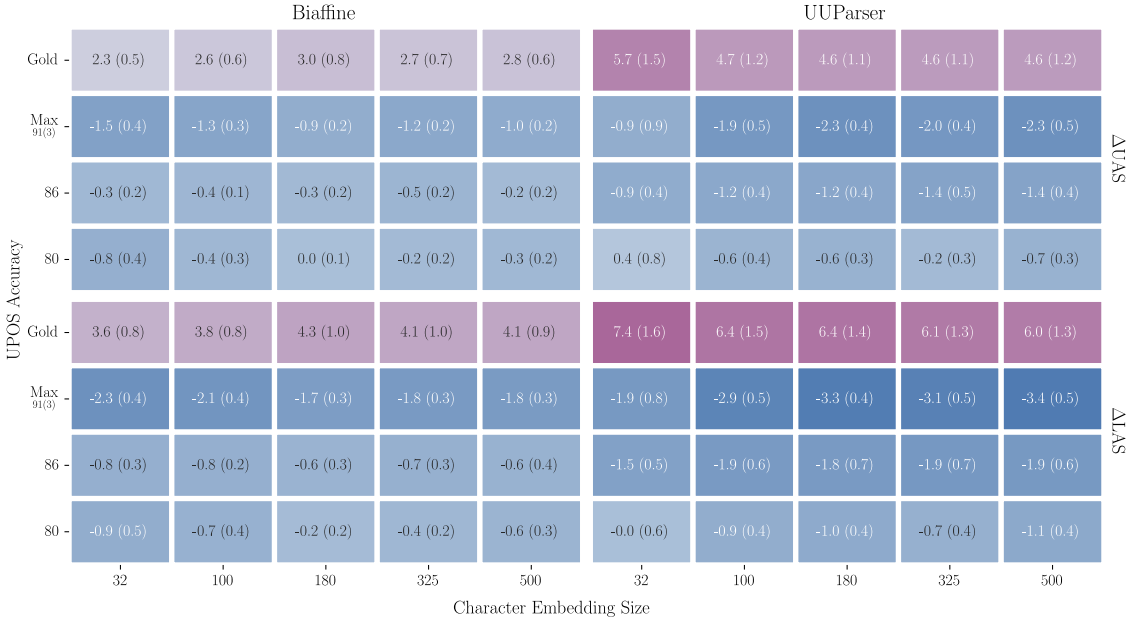


Figure 6.2: Average Δ attachment scores across all treebanks over the relative baseline parsers trained without POS tags for different character embedding sizes and different POS tag accuracies (80%, 86%, max (average of 91%) tagger accuracy for each treebank, and gold).

Experiment 2 The average attachment score differences for Experiment 2 are shown in Figure 6.2. This experiment was initially devised as we anticipated POS tags would have more of a positive effect, especially for higher accuracy taggers, and we wanted to evaluate if having larger character embeddings would offset this. However, as the results of Experiment 1 showed no improvement over not using POS tags at all, this experiment

became a verification of the inutility of predicted POS tags instead. And it is clear that in all contexts where predicted tags are used, no matter what the character embedding size is or what parser is used, predicted POS tags perform worse than not using POS tags at all. The unexpected dip in performance as tagging accuracy increases is even clearer here, as this

	Biaffine		UUParser	
Char	UAS	LAS	UAS	LAS
32	84.0 (5.9)	78.6 (8.7)	77.9 (8.3)	71.9 (10.0)
100	83.9 (6.3)	78.6 (8.9)	79.0 (7.3)	73.0 (9.3)
180	83.4 (6.8)	78.1 (9.4)	79.1 (7.2)	73.1 (9.2)
325	83.6 (6.7)	78.1 (9.5)	79.2 (7.1)	73.3 (8.9)
500	83.6 (6.4)	78.1 (9.1)	79.3 (7.0)	73.4 (8.8)

Table 6.1: Average attachment scores for different character embedding sizes (Char) without POS tags.

trend is consistent across different character embedding sizes and is the case for both parsers. This decrease in performance is even more marked as the performance actually increases when increasing the character embedding size and not using POS tags at all for UUParser, as shown in Table 6.1. This result corroborates one of the many observations from Smith et al. (2018). For the graph-based parser there is a negligible negative impact at higher character embedding sizes. Both parser implementations use a BiLSTM to create the character vector input to the network, so this seems more likely to be a result of the transition-based decoder leveraging features more than the graph-based one. The transition-based parser's ability to leverage POS tags in optimal settings is even clearer in Figure 6.2, as UUParser has twice the improvement using gold tags than that of Biaffine. Also, the impact of predicted POS tags is more pronounced as character embedding sizes increase for UUParser, but for Biaffine there is only a slight tendency to decrease as the character embedding size increases. We

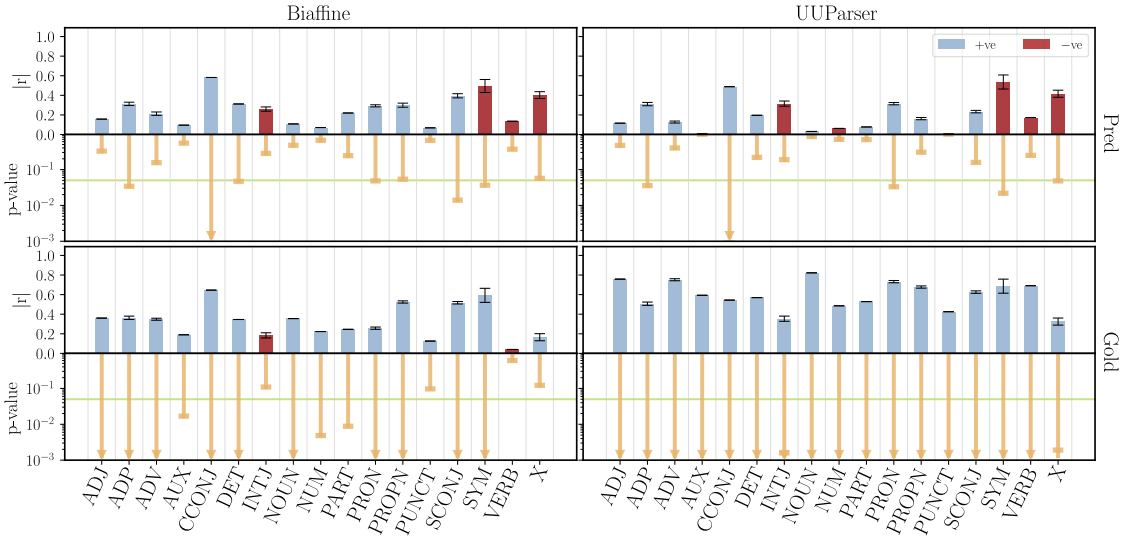


Figure 6.3: Pearson coefficients for the F1-score for separate POS tags and global LAS where positive (+ve) coefficients are shown in blue and negative (-ve) are shown in red. The corresponding p-values are shown below (orange) where an arrow head means the value was below 0.001. Left subplots are for Biaffine parsers, right for UUParsers, top row is for parsers trained with predicted tags, and bottom for parsers trained with gold tags.

show the breakdown for each treebank in Figures 6.18–6.21 in Appendix 6.A.1 where again Tamil is clearly an outlier for UUParser, as it is the only language where any settings with predicted POS tags result in a positive increase (80 POS tag accuracy, character embedding size of 32) and has by far and away the largest increase when using gold tags (a factor of 2

greater than the next best improving language, Wolof, for both UAS and LAS).

Experiment 3 In Figure 6.1, there is a point around 96-98 POS tag accuracy where the parsers outperform the baselines without POS tags. Due to a lack of models in that range, this is just an extrapolation. So we trained parsers with treebanks, listed above in the description of Experiment 3, for which we could obtain high POS tagging accuracy. The results of these parsers are shown in Table 6.2. Only the top two treebanks with the highest tagging accuracy (Catalan AnCora and Japanese GSD) perform better than using no POS tags, and only for UUParser. However, when the performance is below the baseline the difference is marginal. These results are consistent with the extrapolations in Figure 6.1 and suggest a sharp increase in the Δ attachment score slopes when POS tagging accuracy is in the 98-100 range, i.e. that predicted POS tags suddenly start being useful when they are very close to gold POS tags. This suggests that there may be certain tag patterns or contexts that are particularly relevant for parsing, but especially difficult for taggers to learn.

	Biaffine		UUParser		
	UAS	LAS	UAS	LAS	POS_{ACC}
Catalan-AnCora					
Predicted	92.59	89.57	**90.88**	**88.03**	98.26
None	**92.89**	**90.33**	90.82	87.92	n/a
Japanese-GSD					
Predicted	95.02	93.66	**94.56**	**92.94**	97.69
None	**95.12**	**93.54**	94.47	92.74	n/a
Polish-PDB					
Predicted	92.78	89.97	89.25	85.57	97.52
None	**93.64**	**90.94**	**89.32**	**85.60**	n/a
Latin-ITTB					
Predicted	90.92	88.47	86.99	83.99	97.46
None	**91.09**	**88.74**	**87.25**	**84.25**	n/a

Table 6.2: Performance for treebanks with high scoring POS taggers trained with predicted POS tags, compared to the performance on the same treebanks without using POS tags.

6.3.4 Parsing difficulty of POS tags

We then delved deeper by looking at the difficulty of predicting arcs and labels for each POS tag type. The full results are shown in Figures 6.5 and 6.6, where the average differences in score with respect to the baseline model (no POS tags) are given. `X` (the UPOS tag for "other") is consistently difficult across parsers and parser types, except that the loss in UAS for UUParser is much smaller than for Biaffine for both training with predicted and gold tags. However, the only time `X` is consistently better than the baseline model for LAS is when using gold tags at runtime and only with UUParser. Another noticeable feature in these results is that for the max POS predicted accuracy and gold tag parsers for UUParser, `INTJ` (interjection) performs significantly better, both using predicted POS tags and gold tags for training, compared to the lower POS tag accuracy parsers. Beyond this, the performances echo the global scores with respect to the tagging accuracy.

Next, we evaluated the correlations (Pearson coefficient) between tagging accuracy for each POS tag and global parsing performance. For these correlation results and all those that follow, we use the same taggers and parsers from Experiment 1. We only report results for LAS for the sake of space.

Figure 6.3 shows the Pearson coefficient with the corresponding p-value for the correlations between the F1-score for each POS tag and the global LAS score for both Biaffine and UUParser, for both predicted and gold POS tags used in training. Training with gold tags, the accuracy for every tag is positively correlated with parsing performance for UUParser and the correlations are all statistically meaningful. The correlations range from about 0.4 (`INTJ` and `X`) to about 0.8 (`ADJ`, `ADV`, `NOUN`, and `PRON`). For Biaffine, the correlations are much

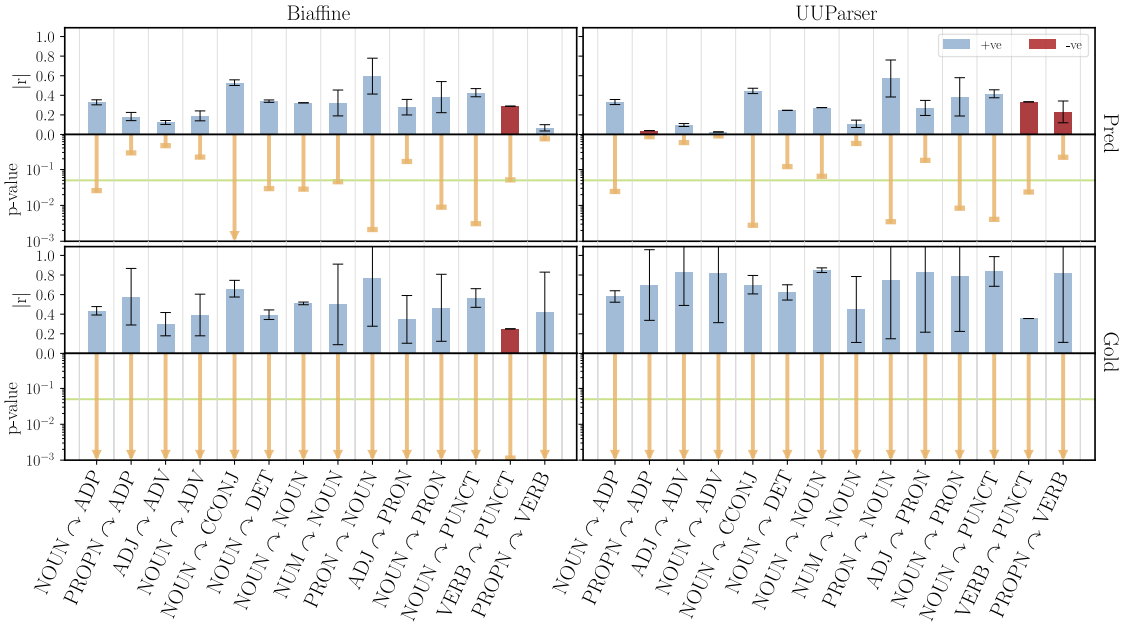


Figure 6.4: Pearson coefficients for the tagging F1-score for child↷head pairs and global LAS where positive (+ve) coefficients are shown in blue and negative (-ve) are shown in red. The corresponding p-values are shown below (orange) where an arrow head means the value was below 0.001. Left subplots are for Biaffine parsers, right for UUParsers, top row is for parsers trained with predicted tags, and bottom for parsers trained with gold tags.

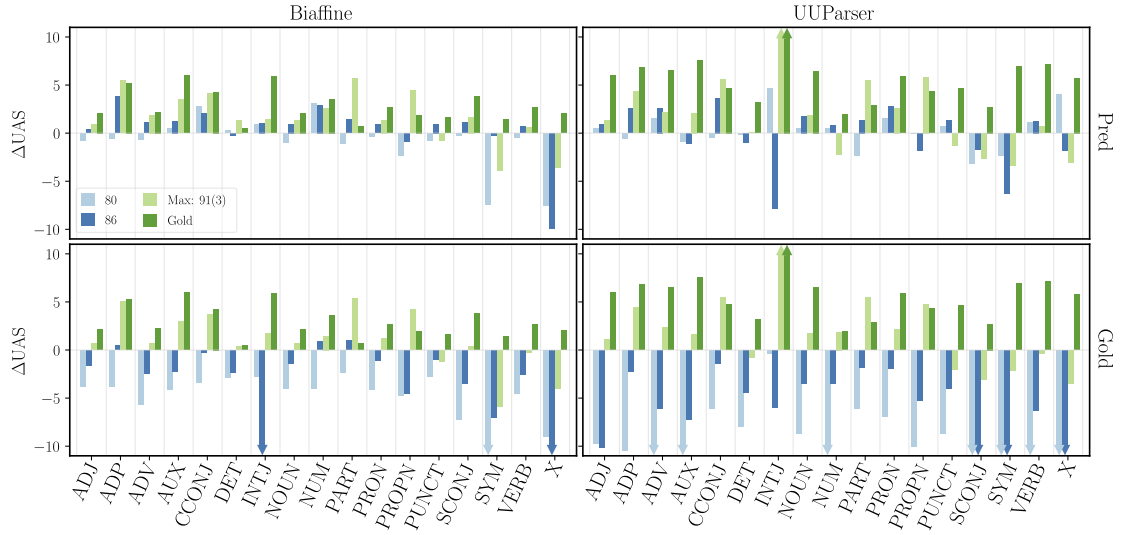


Figure 6.5: Average ΔUAS across all treebanks for models trained with POS tags from taggers of 80, 86, max POS accuracy of 91(3), and with gold tags for each POS tag.

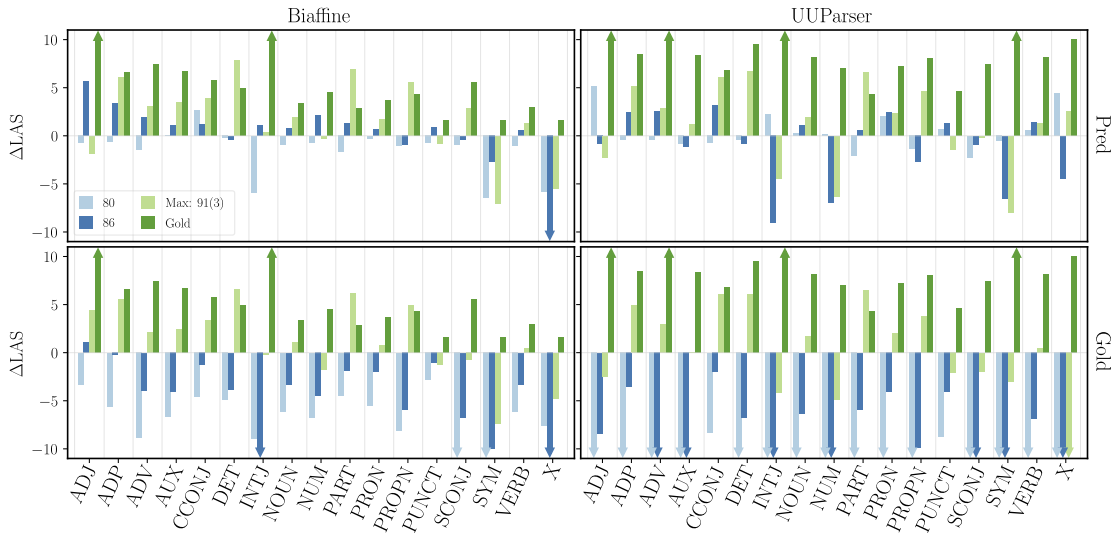


Figure 6.6: Average ΔLAS across all treebanks for models trained with POS tags from taggers of 80, 86, max POS accuracy of 91(3), and with gold tags for each POS tag.

weaker ranging from 0.2 (`AUX`) to 0.6 (`CCONJ`, coordinating conjunction, and `SYM`, symbol) for those which are statistically significant.

For the systems trained with predicted POS tags, the correlations are much weaker for UUParser and only 5 are statistically significant. UUParser and Biaffine have much more similar correlations under these settings, where Biaffine has 2 other tags significantly correlated but its set contains those of UPParser. Of those that are significantly correlated for both, `SYM` and `X` are actually negatively correlated, suggesting that the taggers either fail to generalise or fail to capture certain tagging patterns. A noticeable exception is the `CCONJ` tag which is both strongly correlated (about 0.6 for both) and statistically significant for both parsers. This is likely due to the nature of `conj` relations, where dependents are connected to the conjunct rather than the head of the conjunct (e.g. the second conjoined object of a verb is connected to the first) and so should be parsed differently than if they occurred without a `CCONJ`.

Figure 6.8 shows the correlation for the tagging accuracy of the head for each tag type. Across all systems, there is a correlation for the head of `INTJ` nodes (0.7 for predicted training, 0.5 for gold). This is perhaps due to `INTJ` nodes typically being attached to `VERB` or `NOUN` nodes, and that this narrow context means that the parsers will always look for a node like these and if the correct node is incorrectly tagged, this could disrupt the arc predictions and would be better off without the tagging information. `ADP` (adposition) nodes are similar but with a lower correlation (about 0.4 for all systems). And again this might be due to these nodes occuring in less diverse contexts. `X` nodes are strongly negatively correlated for Biaffine for both gold and predicted training systems (0.6 and and 0.8 respectively) and similarly `SCONJ` (subordinating conjunction) nodes (0.7 and 0.5). Perhaps the diversity of the contexts in which these tags occur makes it difficult for the parser to leverage POS information. `ADV` nodes follow a similar trend, being negatively correlated for 3 of the 4 systems (gold UUParser being the exception), which could also be related to diversity of contexts: adverbs such as *very* never attach to verbs, but to other adverbs, and often the use of `ADV` covers situations where a word doesn't satisfy the definition of another POS tag.

Figure 6.4 shows the correlation of combining the accuracy of POS tags and the tags that govern them with global LAS scores. Only pairs that occur 10 times in 4 treebanks are included. The union of pairs with the highest correlations across all systems are shown (the 10 most highly correlated and statistically significant for each parser). The correlations are

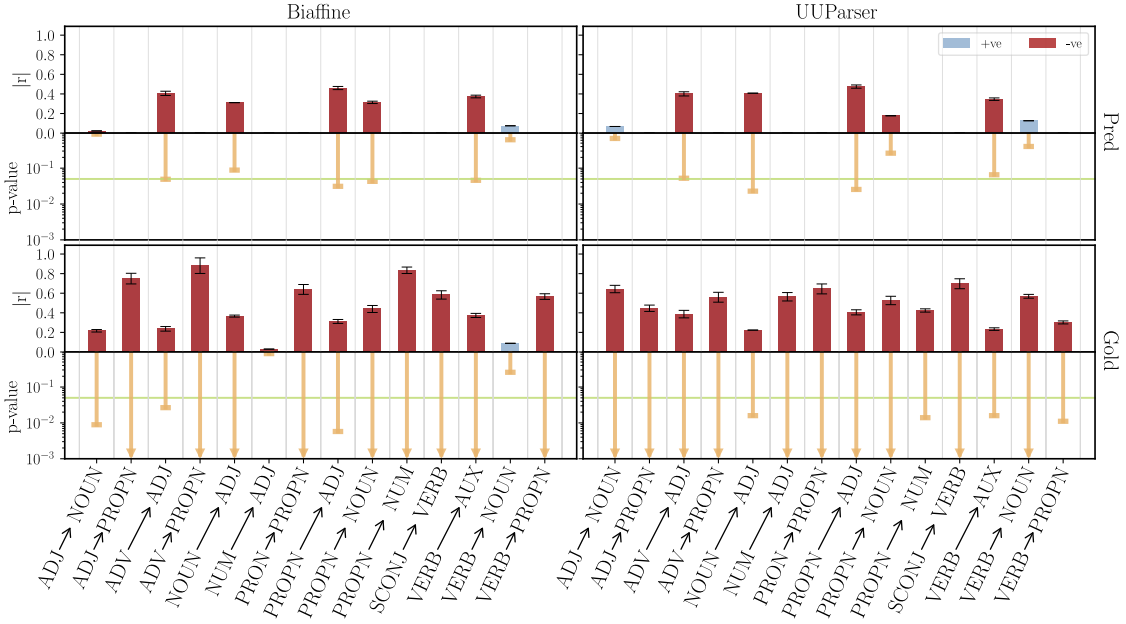


Figure 6.7: Pearson coefficients for the error rate of individual error types $\texttt{POS}_X \rightarrow \texttt{POS}_Y$ and global LAS where positive (+ve) coefficients are shown in blue and negative (-ve) are shown in red. The corresponding p-values are shown below (orange) where an arrow head means the value was below 0.001. Left subplots are for Biaffine parsers, right for UUParsers, top row is for parsers trained with predicted tags, and bottom for parsers trained with gold tags.

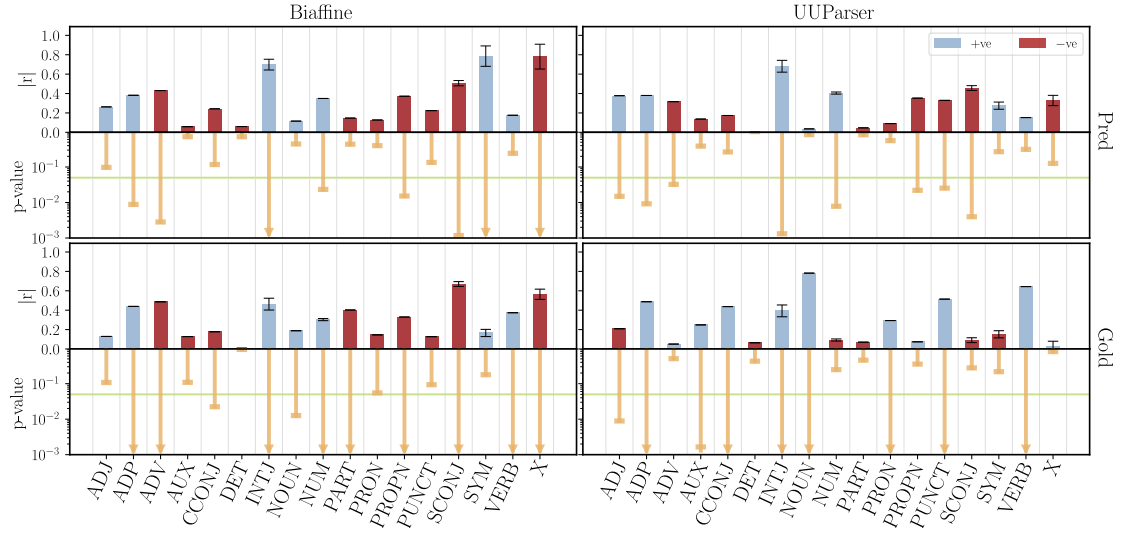


Figure 6.8: Pearson coefficients for the F1-score of the *head* of separate POS tags and global LAS where positive (+ve) coefficients are shown in blue and negative (-ve) are shown in red. The corresponding p-values are shown below (orange) where an arrow head means the value was below 0.001. Left subplots are for Biaffine parsers, right for UUParsers, top row is for parsers trained with predicted tags, and bottom for parsers trained with gold tags.

positive with one exception of PUNCT nodes headed by VERB nodes, which are weakly negative for all systems except gold UUParser. Conversely, PUNCT nodes headed by NOUN nodes have positive correlations for all systems (0.4 for all except gold UUParser which is about 0.8). Other than this, CCONJ nodes headed by NOUN nodes are positively correlated (0.6-0.7) for all systems, which adds to the discussion above regarding CCONJ tags and suggests that it helps more specifically when conjuncts are NOUN nodes.

6.3.5 Dependency distance

Figures 6.22–6.25 in Appendix 6.A.3 show the attachment scores and occurence rates for each POS tag in dependency distance bins. Most tags decrease in performance as the distance increases. Other than NOUN, PUNCT, and VERB, the occurrence of longer-distanced edges are significantly lower than short-distanced ones. Of these, NOUN has a much more significant drop in performance as distance increases across all systems.

Figure 6.9 shows the combinations of POS tag and dependency distance with highest correlation with LAS. CCONJ appears in 8 pairs (out of 24) and appears 3 out of 6 times for the distances of 3 or less. This further supports the findings from above that awareness of CCONJ nodes is especially beneficial. Beyond this, most pairs (19) have distances of 4 or greater which is larger than the mean dependency distance typically observed in natural languages, e.g. it is 3.6 (0.4) averaged over all treebanks in UD v.2.4 using the equation from Liu (2008).

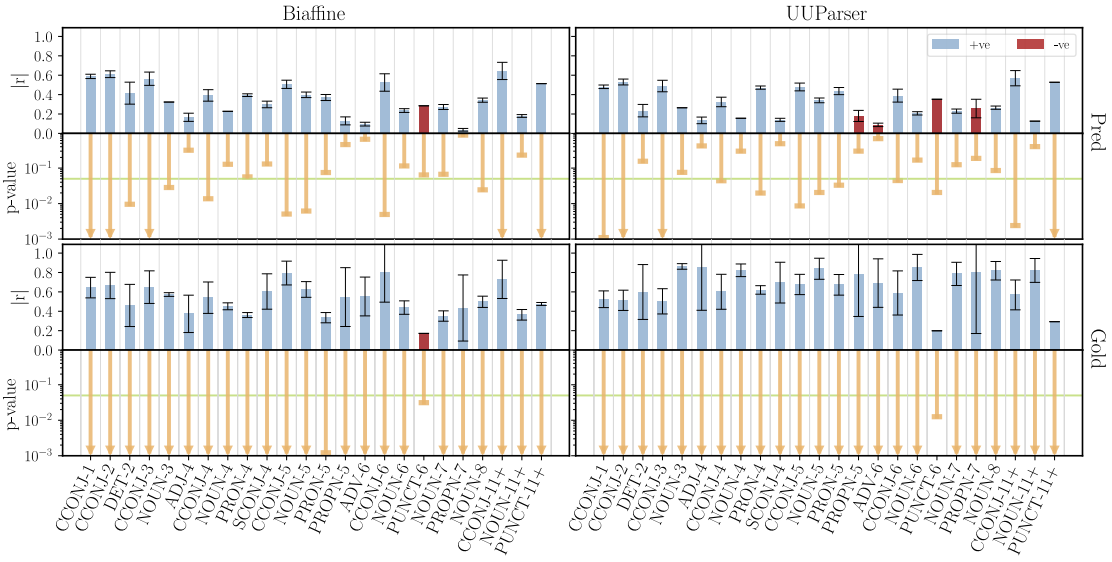


Figure 6.9: Pearson coefficients for the F1-score for individual POS tags with different dependency distances (POS-distance) and global LAS where positive (+ve) coefficients are shown in blue and negative (-ve) are shown in red. The corresponding p-values are shown below (orange) where an arrow head means the value was below 0.001. Left subplots are for Biaffine parsers, right for UUParsers, top row is for parsers trained with predicted tags, and bottom for parsers trained with gold tags.

6.3.6 Error types

Finally, we evaluated which type of tagging errors are the most likely to impact parsing performance. In Figures 6.26–6.29 in Appendix 6.A.3, we show the corresponding attachment scores and counts of each error of tagging a gold tag of $\texttt{POS}_X$ as $\texttt{POS}_Y$ in confusion matrices for both parser types, for training with gold tags and with predicted tags with taggers of accuracy 80, 86, 91(3). We include these to supplement the following analysis and allow for comparisons to other error types that aren't shown. However, we can see that a lower

occurrence rate of errors is associated with lower attachment scores and errors have a larger impact on LAS than UAS.

Figure 6.7 shows the highest correlated and statistically significant tagging errors. Correlations are between the error rate and the global LAS scores. Only error types that occur 10 times in the output of at least two taggers for at least 4 treebanks are included (the 5 most correlated for each parser). This is due to the fact that looking at the correlation between error rates and LAS when an error type rarely occurs will still give *statistically meaningful* correlations, as the absence of stats is one step removed from the correlation calculation. Error types are negatively correlated with parsing performance (the exceptions are those which aren't statistically significant for some systems). Correlations are strongest when training with gold POS tags. For Biaffine they are either much stronger or much weaker than UUParser, e.g. ADJ→PROPN is over 0.8 whereas it is only about 0.5 for UUParser, PROPN→NUM is about 0.8 for Biaffine and about 0.4 for UPParser. In contrast, ADJ→NOUN and ADV→ADJ are only about 0.2 for Biaffine but are about 0.6 and 0.4, respectively, for UUParser.

Two POS tag pairs appear in error types where both directions are observed, PROPN↔ADJ and NOUN↔ADJ for both parsers trained with gold tags. For the former, it appears that qualifiers that refer to nations or groups are often problematic as a similar form or the same one appear as PROPN and ADJ, e.g. *Sunni*, *African*, *Mexican*. For the error type ADJ→PROPN, another issue seems to be the capitalisation of certain words which either appear on their own or with limited punctuation, e.g *Wonderful!*, *Marvelous!*, or refer to something fixed but not quite a named entity, e.g. *Parliamentary elections*, *Perfect Score*. This is the case in English, and we apologise for the Anglo-bias, but the author isn't proficient in the other languages used. However, these errors do occur at a general level. It appears to be similar in Russian (Бургундского - *Burgundy*, Гомельская - *Gomel Region*); Finnish (*Suomalaisen - Finnish*, *eurooppalaisen - European*); and in Hebrew (**איטלקית** - *Italian*, **גרמנית** - *German*). The only language where neither of these error types occur is Wolof as it doesn't have an adjective category (Dione, 2019).

For the other bidirectional error type (NOUN↔ADJ), there appears a similar issue for ADJ→NOUN as ADJ↔PROPN for nations or groups, but NOUN is used instead of PROPN. Beyond this, when a NOUN is incorrectly tagged as an ADJ this occurs 44.7 (14.4)% when it is governed by another NOUN. This is especially prominent for English and Hebrew (65.8% and 64.4% respectively) with the lowest rate occurring for Ancient Greek and Tamil (25.8% and 28.9% respectively). The issue of tagging NOUN tokens governed by another NOUN token is also apparent in Figure 6.4 where this pair has a correlation coefficient of about 0.4 for both Biaffine systems and 0.8 for the gold trained UUParser system (for the predicted POS tag UUParser system, it isn't statistically significant). Again Wolof is an outlier as the error NOUN→ADJ never occurs, presumably because it never has any ADJ tokens to learn.

Only two error types are statistically significant for all systems: ADV→ADJ and PROPN→ADJ, the latter having been discussed above. The former isn't particularly prevalent, occurring with an error rate of 5.8 (5.7)% on average across all languages (except Wolof) with Russian and Tamil having the highest rates (15.5% and 15.0%, respectively) and Chinese and Hebrew having the lowest (0.6% and 1.7%). For English at least, two issues are clear. Words that have the same form when used as an adverb or adjective are commonly mis-tagged as ADJ when they should be ADV, e.g. *more*, *worst*, *better*, and so on. And also when an adverb is used in hyphenated adjectival phrases such as *fully* in *fully-fledged* and *ill* in *ill-advised*.

As Wolof was such an outlier with respect to common tagging errors (with those that impacted parsing performance) we looked at those most common in Wolof. DET→TAG occur more often than average, especially DET→VERB (error rate of 10.0% compared to 2.4(4.1)% for other languages) and DET→PRON (error rate of 13.5% compared to 7.6(6.3)% for other languages). DET→NOUN is also common but similar to the other languages (error rate of

8.0% compared to 7.0(7.0)% for other languages). DET→VERB and DET→NOUN are negatively correlated for Wolof with an average coefficient of −0.85(0.11) across all systems and all with $p < 0.05$. Clearly, further language-specific analyses are needed.

6.3.7 Summary

We have evaluated the impact POS tag accuracy has on parsing performance for leading graph- and transition-based parsers across a diverse range of UD treebanks, highlighting the stark difference between using predicted POS tags and gold POS tags at runtime. We observed a non-linear increase in performance when using gold tags, suggesting they are somehow exceptional, i.e. precisely the tag patterns that not even the most accurate taggers can correctly predict (the last 2-3 percentage points towards 100% accuracy) seem to be the most important for parsing. This could be due to the parsers implicitly learning POS tag information, in such a way that the taggers learn nothing new to contribute or not enough to avoid a loss in performance due to the errors disrupting what the parsers have learnt. Our analysis also shows that practitioners should evaluate the efficacy of using predicted tags for a given system or language. We have also analysed what aspects of erroneous tagging predictions have the greatest impact and correlation to parsing performance. We observed some global trends, like the importance of CCONJ, but also language-specific issues which highlight the need to evaluate the usefulness of POS tags per language. The results also suggest that using a subset of POS tags might be effective.

6.4 What Taggers Fail to Learn, Parsers Need the Most

We present an error analysis of neural UPOS taggers to evaluate why using gold tags has such a large positive contribution to parsing performance while using predicted UPOS either harms performance or offers a negligible improvement. We also evaluate what neural dependency parsers implicitly learn about word types and how this relates to the errors taggers make, to explain the minimal impact using predicted tags has on parsers. We then masked UPOS tags based on errors made by taggers to test in what way predicted UPOS tags are detrimental if parsers are implicitly aware of UPOS tags.

6.4.1 Methodology

We performed two experiments. The first was an attempt to compare what biaffine parsers learn about UPOS tags by finetuning them with tagging information and comparing their errors with those from normally trained UPOS taggers. The second experiment attempted to evaluate the impact tagging errors have by either masking errors or using the gold tags for erroneously predicted tags and masking all other tags.

Data We took a subset of UD v2.6 treebanks consisting of 11 languages all of which are from different language families (Zeman et al., 2020): Arabic PADT (ar), Basque BDT (eu), Finnish TDT (fi), Indonesian GSD (id), Irish IDT (ga), Japanese GSD (ja), Korean Kaist (ko), Tamil TTB (ta), Turkish IMST (tr), Vietnamese VTB (vi), and Wolof WTB (wo). We used pre-trained word embeddings from fastText (for Wolof we had to use the previous Wiki version) (Bojanowski et al., 2017; Grave et al., 2018). We compressed the word embeddings to 100 dimensions with PCA.

Experiment 1 - error crossover We trained parsers and taggers on the subset of UD treebanks described above. We then took the parser network and replaced the biaffine structure with an MLP to predict UPOS tags. We froze the network except for the MLP and finetuned the MLP with one epoch of learning. We repeated this for the tagger networks (replacing their MLP with a randomly initialised MLP) to validate this finetuning procedure. We then compared the tagging errors of both the finetuned parser and the original taggers. We also undertook an analysis of the errors from the normal taggers.

Experiment 2 - masked tags We then used the output from the taggers from Experiment 1 to train different parsers. We trained parsers using all the predicted tags, using only the gold tags the taggers failed to predict (for both the standard taggers and finetuned parsers), using predicted tags from the standard taggers but masking the errors, and training with all gold tags. We also trained parsers with no tags as a baseline.

	Tagger	Tagger-FT	Parser
Arabic	96.71	96.52	93.73
Basque	95.35	95.18	88.09
Finnish	96.92	96.62	92.24
Indonesian	93.72	93.79	91.98
Irish	92.84	92.80	88.24
Japanese	97.94	97.85	92.80
Korean	95.09	94.26	86.93
Tamil	89.29	87.28	75.41
Turkey	95.10	94.98	86.14
Vietnamese	87.85	87.63	83.40
Wolof	93.85	93.79	85.81

Table 6.3: Tagging accuracies for tagger trained normally, "finetuning" a newly initialised MLP for the trained taggers, and for parsers finetuned to predict tags.

Network details Both the taggers and parsers use pre-trained word embeddings and character embeddings. The parsers use UPOS tag embeddings as specified in the experimental details. The character and tag embeddings are randomly initialised. The parsers consist of the embedding layer followed by BiLSTMs layers and then a biaffine mechanism (Dozat and Manning, 2017). The taggers are similar but with an MLP following the BiLSTMs instead. We ran a nominal hyperparameter search using fi, ga, tr, and wo and using their respective development data. This resulted in 3 BiLSTM layers with 200 nodes, 100 dimensions for each embedding type with 100 dimension input to the character LSTM. The arc MLP of the biaffine structure had 100 dimensions, 50 for the relation MLP. Dropout was 0.33 for all layers. Learning rate was 2×10^{-3}, β_1 and β_2 were 0.9, batch size was 30, and we trained both taggers and parsers for 200 epochs but with early stopping if no improvement was seen after 20 epochs. Models were selected based on the performance on the development set.

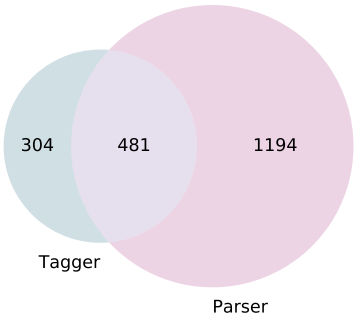


Figure 6.10: Average union of tagging errors for parser with finetuning and fully-trained tagger (stds: 159 for tagger error, 715 for parser, and 242 for union).

6.4.2 Results and discussion

Experiment 1 - error crossover Table 6.3 shows the tagging performance for the normally trained taggers, the re-finetuned taggers, and the finetuned parser taggers. The re-finetuned taggers achieve relatively similar performance to the original taggers, which suggests that this procedure does allow us to develop a decoder that captures what the

	All	Open	Closed	Other
Tagger	8,637	6,434	1,867	336
Parser	18,426	15,181	2,816	429
Total	171,373	101,965	46,362	23,046

Table 6.4: Error counts per word type class of gold tag.

	Error Types					Errors	Tokens
ar	noun→x 197	x→noun 139	noun→adj 108	adj→x 78	adj→noun 60	931	28.3K
eu	propn→noun 145	verb→aux 113	noun→adj 101	aux→verb 100	adj→noun 94	1134	24.4K
fi	propn→noun 56	noun→propn 53	noun→adj 43	adj→noun 39	noun→verb 37	649	21.1K
id	propn→noun 147	noun→propn 92	adj→noun 47	noun→adj 34	verb→noun 23	740	11.8K
ga	propn→noun 184	noun→propn 53	noun→adj 53	adj→noun 38	noun→pron 36	724	10.1K
ja	noun→adv 52	propn→noun 24	noun→adj 22	adj→noun 22	aux→verb 20	269	13.0K
ko	noun→propn 252	propn→noun 145	verb→adj 133	aux→verb 78	cconj→sconj 75	1394	28.4K
ta	noun→propn 24	aux→verb 22	propn→noun 17	noun→verb 12	adj→adp 12	213	2.0K
tr	noun→adj 54	propn→noun 52	noun→verb 37	noun→propn 35	adv→adj 31	491	10.0K
vi	noun→verb 201	verb→noun 152	noun→adj 151	verb→adj 140	verb→x 83	1452	12.0K
wo	noun→propn 71	verb→noun 57	pron→det 46	noun→verb 38	verb→aux 30	640	10.4K

Table 6.5: Top 5 most common errors and their number of occurrences for each treebank. Also shown are the total number of errors and token count for each treebank.

BiLSTM and embedding layers learn about UPOS tags without adding new information. Clearly more training would likely improve the finetuned parsers, but it would be less clear if that would be extracting information the parser previously learnt or adding more information via the weights of the MLP.

	Tagger	Parser
PUNCT	99.94	99.93
SYM	97.83	0.00
X	76.37	54.51
ADJ	87.98	74.98
ADV	93.94	89.97
INTJ	40.91	0.00
NOUN	95.49	94.63
PROPN	90.21	57.49
VERB	94.80	94.05
ADP	97.77	94.14
AUX	96.37	93.65
CCONJ	96.30	94.29
DET	94.73	86.88
NUM	93.96	78.12
PART	90.49	76.88
PRON	96.31	72.46
SCONJ	93.15	91.22

Table 6.6: F1-score for separate tags clustered by word type class with "Other" at the top, "Open" in the middle, and "Closed" at the bottom for all tokens in the collection of treebanks used.

Figure 6.10 shows the average cross-over of specific error occurrences for the two systems where only 38% of the tagger's errors don't occur for the parser. Table 6.4 shows the breakdown of errors from each system by word type class for all treebanks. The ratio of the errors is substantially different for each class: 0.42 for *open*, 0.66 for *closed*, 0.78 for *other*. This perhaps suggests that the parser has a tendency to learn more syntactical fixed word types than open types. Table 6.6 shows the F1-score for each UPOS for both systems. For the most part the parser is pretty close to the tagger for open class tags, except for `INTJ` which the parser never predicts and for `PROPN` (32.7 less for the parser) and `ADJ` (13.0 less).

Table 6.5 shows the top 5 most common errors per treebank for the normal taggers where PROPN appears in 15 error types and ADJ appears in 19 out of 55. This prevalence combined with the parsers' poor performance for these tags suggests that errors containing these tags are especially impactful for parsers when using predicted UPOS. However, it could also be that the parsers perform poorly on predicting PROPN tags as they occur in similar syntactic roles as NOUN tokens and as such isn't as important for syntactic analysis.

For the closed class type tags, again the parser performs similarly to the tagger but a few points less except for NUM, PART, and PRON with drops for parser scores of 15.8, 13.6, and 23.9, respectively. However, of these 3 tags, only PRON appears in the most common errors and only twice. The most common tag to appear in an error is NOUN occurring 41 times, but there is less than one point in difference between the tagger's performance and the parser's for NOUN. Of these 41 appearances, 14 co-occur with ADJ and 15 with PROPN with a fairly even split of mis-tagging NOUN as either of these tags or the other way around. So generally NOUN tokens are fairly easy to tag but the times where the tagger fails is typically where there is confusion with ADJ and PROPN tags. Figure 6.11 shows statistical metrics of

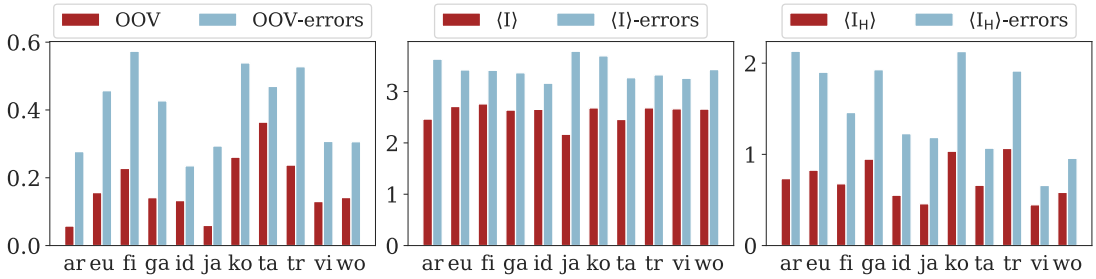


Figure 6.11: Measurements of all tags (red) and error (blue) tags for OOV proportion, POS bigram surprisal (⟨I⟩, ⟨I⟩-errors), and head POS and relation surprisal (⟨I_H⟩, ⟨I_H⟩-errors).

the taggers' errors. First, we show the proportion of out-of-vocabulary (OOV) word forms for all tokens and also the tokens where the tagger makes an error. Consistently across all treebanks the OOV proportion is considerably higher for tokens erroneously tagged. Second, we report the mean UPOS surprisal. For a given UPOS tag, θ_n for token n, the surprisal of that UPOS tag in a given context, c_k is given as:

$$I(\theta_n) = -\log_2 p(\theta_n | c_k) \tag{6.1}$$

where we use a bigram context:

$$c_k = (\theta_{n-2}, \theta_{n-1}) \tag{6.2}$$

Then the mean surprisal, $\langle I \rangle$, over a sample of tokens is given as:

$$\langle I \rangle = \frac{1}{N} \sum_{n \in N} I(\theta_n) \tag{6.3}$$

where N is the number of tokens in the sample. Again, the mean tag surprisal is substantially different across all treebanks for the tokens where the tagger makes a mistake in comparison to the average over the entire treebank. Finally, we report the mean surprisal of UPOS but with the context of its head's tag and the syntactic relation joining the two tokens, such that c_k is defined as:

$$c_k = (\theta_{head}, rel) \tag{6.4}$$

The difference between the error sub-sample and the whole treebank is starker for the head-relation surprisal, suggesting that the tagger struggles more when the syntactic structure is uncommon.

	None	Pred.	M¬E$_T$	M¬E$_P$	M∀E$_T$	Gold
ar	83.29	82.87	84.17	84.06	84.45	84.73
eu	81.12	81.14	82.33	82.62	83.13	84.45
fi	85.96	86.04	86.88	87.09	87.61	88.80
id	79.04	78.95	82.20	82.69	81.08	82.95
ga	76.13	76.57	76.62	76.65	77.46	77.90
ja	93.15	92.72	94.41	94.38	94.39	95.30
ko	85.40	85.86	87.53	87.82	87.44	88.52
ta	65.61	64.50	70.24	66.67	66.01	71.95
tr	66.67	67.68	67.62	67.66	67.84	68.86
vi	58.43	60.09	65.42	66.75	65.18	70.87
wo	77.87	78.49	82.03	81.39	81.11	85.41
avg	77.52	77.72	79.95	79.80	79.61	81.79

Table 6.7: LAS parser performance with no tags (None), with predicted tags (Pred), gold tags but with all tags masked except those the respective taggers predicted wrong (M¬E$_T$), similarly for the errors from the fine-tuned parser (M¬E$_P$), masking the errors from the tagger (M∀E$_T$), and finally using all gold tags.

Experiment 2 - masked tags Table 6.7 shows the labelled attachment scores for parsers with varying types of UPOS input. First we use the predicted output from the normal taggers from Experiment 1 (Pred) and unlike Anderson and Gómez-Rodríguez (2020c) we observe a nominal increase over using no UPOS tags. However, using predicted tags isn't universally beneficial. Arabic, Indonesian, Japanese, and Tamil all perform better with no tags.

We then tried using gold tags but masking the tags that the taggers correctly predicted to test if the erroneous tags are particularly useful. We did this for the normal taggers (M¬E$_T$) and also for the finetuned parsers (M¬E$_P$). The average increase for both is about 2.5 over the no tag baseline and over 2 points better than using predicted tags. Also, the improvement is universal with at least a small increase in performance over using predicted UPOS tags. Interestingly the smaller set from the tagger outperforms the larger set from the parser by 0.15, suggests that what both the taggers and the parsers fail to capture is more important than the errors unique to the parsers. We then masked the errors from the taggers (M∀E$_T$) to test if avoiding adding errors would still be beneficial. The performance is almost 2 points better than using the predicted tags and again an increase is observed for all treebanks. This could be of use, as it is easy to envisage a tagger which learns to predict tags when a prediction is clear and to predict nothing when the probability is low. Finally, using gold tags is nearly 2 points better on average than the best masked tag model, which suggests that to fully utilise the information in the final few percentage that taggers miss, the full set of easy to predict tags are needed.

6.4.3 Summary

We have presented results which suggest that parsers do learn something of word types and that what taggers fail to learn is needed to augment that knowledge. We have evaluated the nature of typical tagging errors for a diverse subset of UD treebanks and highlighted consistent error types and also what statistical features they have compared to the average measurement across all tokens in a treebank. We have shown that it would be more beneficial to implement taggers to not only predict tags but also decide when to do so, as the errors undermine anything gained from using predicted tags for dependency parsers.

6.5 *A Falta de Pan, Buenas Son Tortas*[4]

The results in Section 6.3 suggested that smaller treebanks might be able to directly utilise less accurate UPOS tags. This could be potentially useful because, as mentioned above, low resource tagging performance is still very weak. We evaluate this further by analysing the impact of tagging accuracy on UD parsing in low resource contexts, with regards to the amount of data available to train taggers and parsers.

We evaluate the efficacy of predicted UPOS tags as input features for dependency parsers in lower resource settings to evaluate how treebank size affects the impact tagging accuracy has on parsing performance. We do this for real low resource universal dependency treebanks, artificially low resource data with varying treebank sizes, and for very small treebanks with varying amounts of augmented data. We find that predicted UPOS tags are somewhat helpful for low resource treebanks, especially when fewer fully-annotated trees are available. We also find that this positive impact diminishes as the amount of data increases.

6.5.1 Methodology

We performed three experiments. The first is an evaluation of predicted tags as features for biaffine parsers for real low resource treebanks. It also includes parsers trained with UPOS tagging as an auxiliary task similar to the experiments in Zhang et al. (2020b). The second experiment evaluates the impact of different tagging accuracies on different dataset sizes using artificial low resource treebanks by sampling from high resource treebanks. The last experiment utilises a data augmentation technique to investigate the efficacy of predicted UPOS tags for very small treebanks (∼20 sentences) when augmented with varying amounts of data.

Low resource data We take all UD v2.6 treebanks (Zeman et al., 2020) with less than 750 sentences in both its training dataset and development dataset (if available). We cluster these treebanks into two groups, very low with less than 50 sentences and low with less than 750. The very low resource treebanks consist of Buryat BDT (bxr), Kazakh KTB (kk), Kurmanji MG (kmr), Livvi KKPP (olo), and Upper Sorbian UFAL (hsb). The low resource set is made up of Belarusian HSE (be), Galician TreeGal (gl), Lithuanian HSE (lt), Marathi UFAL (mr), Old Russian RNC (orv), Tamil TTB (ta), and Welsh CCG (cy). We combined the training and development data (when available) to then split them 80|20. The statistics for the resulting splits are shown in Table 6.8. We use the original test data for analysis.

Artificial low resource data We use Indonesian GSD (id), Irish IDT (ga), Japanese GSD (ja), and Wolof WTB (wo) to create artificially low resource treebanks. For each we take a sample of 100, 232, and 541 sentences from the training and development data. These samples are then split 80|20 for training and development data. We do this three times for each treebank size so we have multiple samples to verify our results. We use the original test data for analysis.

Augmented data For the experiment using augmented data we use a subset of the smallest treebanks, namely Kazakh, Kurmanji, and Upper Sorbian. We then generate data using the subtree swapping data augmentation technique of Dehouck and Gómez-Rodríguez (2020). We generate 10, 25, and 50 trees for each and we then split them 80|20. We do this three times for each number of generated trees. We use the original test data for analysis.

Subtree swapping We gather all the sub-trees with a continuous span which has a `NOUN`, `VERB`, `ADJ` or `PROPN` as its root node. Other UPOS tags are not used due the likelihood of generating ungrammatical structures. With regards to the permitted relation of the root nodes, we consider all core arguments, all nominal dependents, and most non-core

[4] *Lacking yeast-proven bread, a flatbread alternative will suffice*, i.e. if you can't get more fully-annotated dependency trees, annotating UPOS tags can still be helpful.

dependents (excluding `discourse`, `expl` and `dislocated`). Then given a tree, we swap one of its sub-trees with one from another tree given that their respective roots have the same UPOS tag, dependency relation and morphological features and given that the sub-trees are lexically different. We repeat the process a second time using a third tree. During this second swap, we do not allow the previously swapped subtree to be altered again so as to avoid redundancy. For a more detailed description of this process see Dehouck and Gómez-Rodríguez (2020). We create all possible trees generated from the three original trees given the constraints described above, repeat this for each triplet of trees, and finally take a sample from this set of augmented data.

	Train Sentences	Train Tokens	Dev Sentences	Dev Tokens
bxr	15	120	4	33
kk	24	395	7	134
kmr	16	192	4	50
olo	15	114	4	30
hsb	18	310	5	150
be	307	6,441	77	1,449
gl	480	12,317	120	3,119
lt	166	3,444	42	852
mr	335	2,751	84	686
orv	256	8,253	64	1903
ta	383	6,082	96	1,254
cy	491	10,719	123	2,616

Table 6.8: Number of trees in training and development splits as considered low resource UD treebanks.

Controlling UPOS accuracy For each treebank size and split for the artificial low resource treebanks we trained taggers with varying accuracies (60, 66, 72, 78, 85, 89). We allowed a small window around the accuracy for each bin of ±0.25. Following a similar methodology to that used in Section 6.3 to obtain taggers with varying accuracies, we train the taggers as normal and save models when they reach a desired accuracy. We then train parsers using predicted tags from each of the taggers and use predicted tags at inference. For the data augmentation experiment we used accuracy bins of 41, 44, 48, and 51.

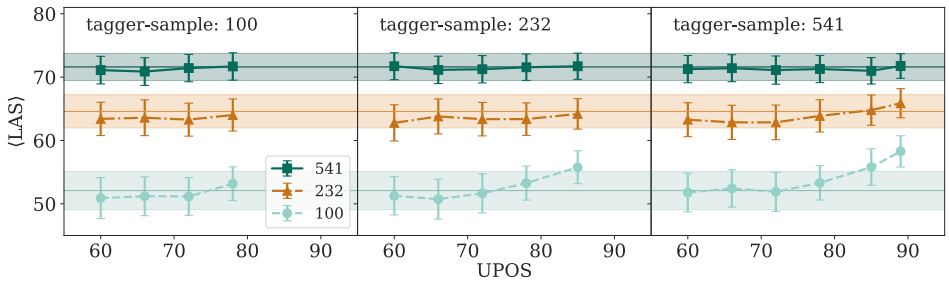


Figure 6.12: Impact of tagging accuracy for varying amounts of data for both taggers and parsers using artificial low resource data. The standard error of UPOS accuracy is not shown as it is very small (< 0.1% relative error for all bins). Horizontal lines and corresponding shaded area show the mean parsing performance and the standard error for the baseline parsers trained without UPOS tags.

Network details Both the taggers and parsers use word embeddings and character embeddings. The parsers use UPOS tag embeddings except for the MTL setup and the baseline models without tags. The embeddings are randomly initialised. The parsers consist of the embedding layer followed by BiLSTM layers and then a biaffine mechanism. The taggers are similar but with an MLP following the BiLSTMs instead. We ran a nominal

hyperparameter search evaluated on the development data of Irish and Wolof. This resulted in 3 BiLSTM layers with 200 nodes, 100 dimensions for each embedding type with 100 dimensions for input to the character LSTM. The arc MLP of the biaffine structure has 100 dimensions, whereas the relation MLP has 50.

6.5.2 Results and discussion

	UPOS		LAS			
	Single	Multi	None	Pred	Gold	Multi
bxr	48.72	48.34	10.45	12.36	20.31	14.41
kk	53.37	52.14	22.48	21.63	36.66	23.50
kmr	50.56	53.73	19.16	18.31	35.54	21.58
olo	37.84	37.37	9.74	10.89	17.54	7.59
hsb	53.44	47.28	18.36	20.03	41.88	14.66
avg	48.79	47.77	16.04	16.64	30.39	16.25

(a) Very low resource: less than 50 sentences.

	UPOS		LAS			
	Single	Multi	None	Pred	Gold	Multi
be	92.82	87.29	61.82	64.91	68.87	62.28
gl	93.54	88.56	70.60	72.73	79.06	70.54
lt	79.25	71.51	37.17	35.94	48.30	38.96
mr	80.58	76.46	57.04	58.74	64.32	56.31
orv	87.77	81.60	49.53	51.34	60.24	50.33
ta	86.88	79.23	63.85	62.75	74.31	63.15
cy	91.77	86.41	72.10	72.93	80.71	73.00
avg	85.89	77.77	55.24	56.52	64.13	55.10

(b) Low resource: less than 750 sentences.

Table 6.9: Performance of different low resource parsers: using predicted UPOS tags as features (Pred), multi-task system where tagging is an auxiliary task to parsing (Multi), using gold UPOS tags as features (Gold), and without using UPOS tags as features (None). The accuracies of the predicted UPOS tags (Single) and that of the multi-task (Multi) are also reported.

Table 6.9 shows the real low resource treebank results. Table 6.9a shows the results for the treebanks with less than 50 sentences. The performance is very low across the board so it is difficult to draw any substantial conclusions, however, using gold tags has a large impact over not using any, almost doubling the label attachment score. Also, using predicted tags does result in an increase on average, but Kazakh and Kurmanji lose almost a point. Further those two treebanks and also Buryat have reasonable gains when using the multi-task framework. The average multi-task score is strongly affected by the large drop seen for Upper Sorbian, which also suffers with respect to tagging accuracy when using the multi-task setup.

Table 6.9b shows the results for the low resource treebanks with less than 750 sentences. On average using predicted UPOS tags achieves a sizeable increase over not using any tags of about 1.2, despite the average tagging accuracy only being 85.89%. This suggests that in a lower resource setting the tagging accuracy doesn't have to be quite so high as is needed for high resource settings. Increases in performance are seen for all treebanks except Lithuanian and Tamil. While Lithuanian has the second lowest tagging score, Tamil has a fairly high score, so it seems that the accuracy needed is somewhat language-specific or at the very least data-dependent. The difference for the treebanks in Table 6.9b is almost 9 points higher for using gold tags. The multi-task performance is about 1.4 points less than using predicted tags on average. However, Lithuanian and Tamil obtain an increase in performance using the multi-task system in comparison to using predicted tags.

Figure 6.12 shows the average LAS performance for the parsers trained with the artificial low resource data. When the parsers have sufficient data, using UPOS tags doesn't offer

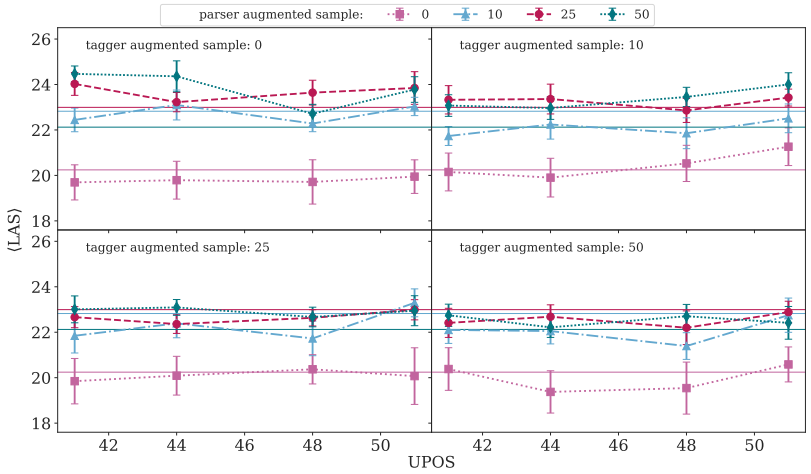


Figure 6.13: Impact of tagging accuracy for varying amounts of data for both taggers and parsers using augmented data (0, 10, 25, and 50 augmented trees) on top of the original gold data. The standard error of UPOS accuracy is not shown as it is very small ($< 0.1\%$ relative error for all bins). Horizontal lines show the mean parsing performance for the baseline parsers trained without UPOS tags (standard error not shown due to too much overlap between augmented data sample sizes).

any improvement in performance. For the parsers trained with 232 samples, there is a slight upward trend when using tags predicted from taggers trained with 541 samples. The improvement increases with respect to UPOS tag accuracy and exceeds the performance of the parsers trained with no UPOS tags. The most noticeable improvement is for the parsers trained with only 100 samples. The impact of UPOS accuracy is clearer as the tagger sample size increases as higher accuracies can be obtained. The best performance is with the most accurate taggers (89%).

This is a potentially useful finding if annotators have little time, as annotating UPOS tags is much less time-sensitive and can help improve parsing performance if a limited number of tree-annotated sentences are available. However, taking parsers using only 100 fully-annotated training sentences as a baseline, the average performance using 232 parsed sentences without UPOS tags is over 10 points higher, whereas the increase gained training the taggers with 541 tagged sentences is only 5 points. So it is clear that if time permits such that annotators can increase the number of tree annotations, they will likely prove to be more useful. But UPOS tags could be obtained using projection methods and/or active learning techniques (Baldridge and Palmer, 2009; Das and Petrov, 2011; Garrette et al., 2013; Täckström et al., 2013). Also, multilingual projection methods could be used, but they typically generate trees as well as POS tags (Agić et al., 2016; Johannsen et al., 2016).

Figure 6.13 shows the impact of predicted UPOS accuracy when using data generated with subtree swapping augmentation. The first result worth noting is that the augmented data increases performance in this very low resource context. Across the board, the best performing parsers using augmented data outperform the parsers trained only on gold data by 3-6 points which corroborates the findings in previous work. However, it appears that there is a limit to how much augmented data helps as the performance of the parsers which use 25 and 50 augmented instances is similar.

It also appears that this upper limit is even lower for training taggers with the best performance coming when using predicted tags from taggers utilising only 10 augmented samples or none at all. Using more invariably hurts performance no matter what accuracy the taggers obtained, as can be seen in the subplots showing the performance for parsers trained with predicted tags from taggers using 25 and 50 augmented samples. Also, there is no clear trend showing the impact of UPOS accuracy in this very low resource context.

6.5.3 Summary

We have presented results which suggest that lower accuracy taggers can still be beneficial when little data is available for training parsers, but this requires a high ratio of UPOS annotated data to tree annotated data. Experiments using artificial low resource treebanks highlight that this utility diminishes if the number of samples reaches a fairly small amount. We have also shown that very small treebanks can benefit from augmented data and utilise predicted UPOS tags even when they come from taggers with very low accuracy.

6.6 Conclusion

In Section 6.3, we undertook a series of controlled experiments where we altered the accuracy of POS taggers. We observed a linear trend for two modern parsing systems of different types and found that using predicted tags was almost exclusively worse than using no tags. Further, we observed a non-linear increase in parsing performance when using gold POS tags suggesting some sort of exceptionality. We also obtained tentative results that suggest smaller treebanks can still benefit from using predicted tags from inaccurate taggers.

In Section 6.4, we investigated this exceptionality by analysing what parsers inherently learn about word types and how this corresponds to what taggers fail to predict. We found that there was a strong cross-over between what parsers fail to learn implicitly and what taggers fail to predict. We also highlighted consistent error types and contexts across a subset of UD treebanks. And obtained results that suggest a more complex tagger would be beneficial if it could learn when it should make predictions and when not to, so as to avoid sending erroneous signals to the parsers. The set of treebanks used was different than that used in Section 6.3 and in this analysis we did observe a moderate increase in performance when using predicted tags over not using them at all (0.2 LAS).

Finally, in Section 6.5, we investigated whether low-accuracy POS taggers are still useful in low-resource contexts. We found that they do offer a small increase in parsing performance ($\approx$1 LAS) and more specifically that we can increase parsing performance if we have more data that is solely annotated with POS tags.

In general the work in this chapter has highlighted that the efficacy of predicted POS tags should not be taken for granted and that when developing parsers it is necessary to evaluate whether using them is beneficial or not. This is likely to be language and resource dependent.

6.A Appendix

6.A.1 Treebank performances for experiment 1 in Section 6.3

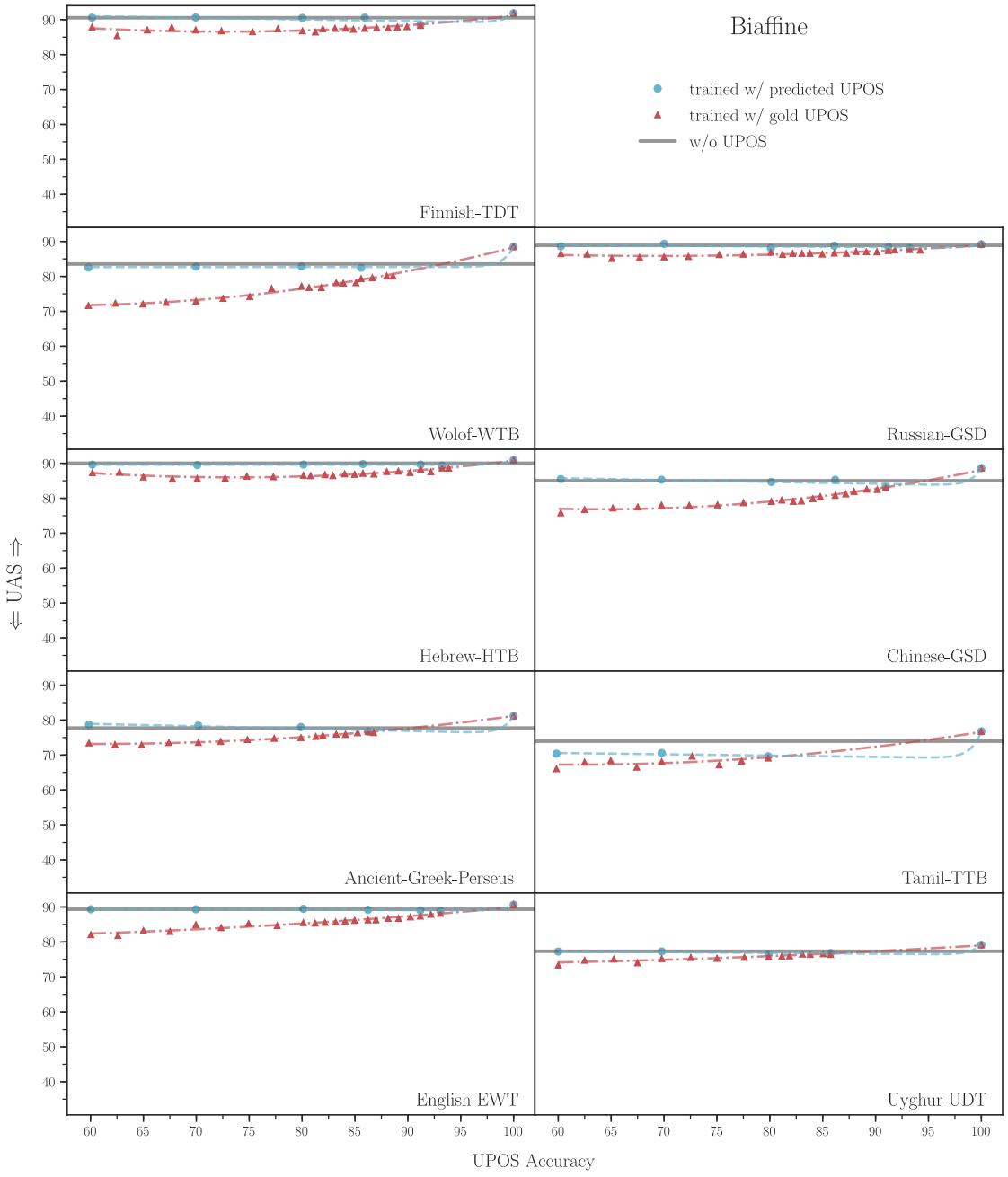


Figure 6.14: UAS for each treebank for Biaffine training with gold (red, triangles) and predicted (blue, circles) POS tags. Baseline parser trained without POS tags is shown in grey.

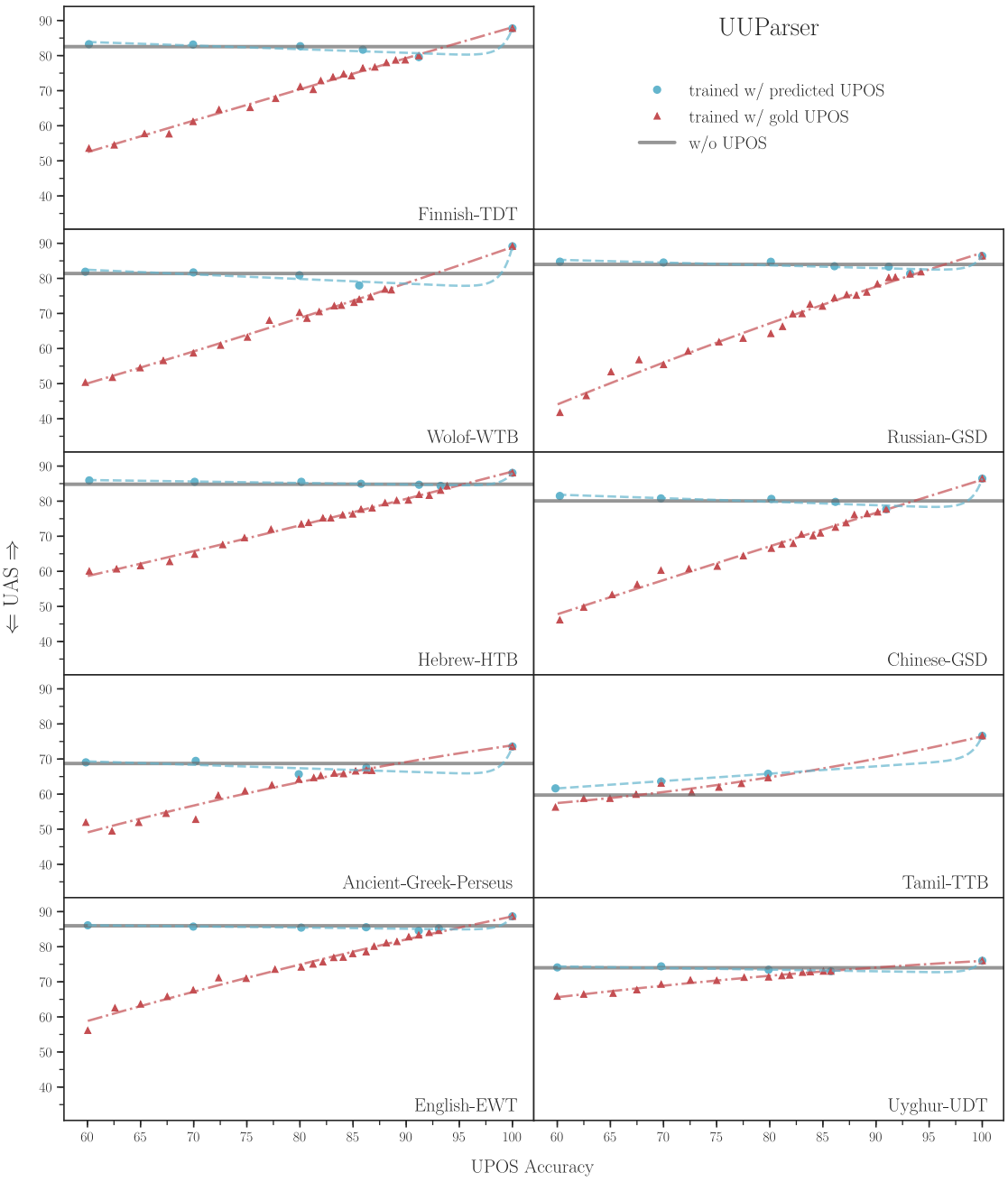


Figure 6.15: UAS for each treebank for UUParser training with gold (red, triangles) and predicted (blue, circles) POS tags. Baseline parser trained without POS tags is shown in grey.

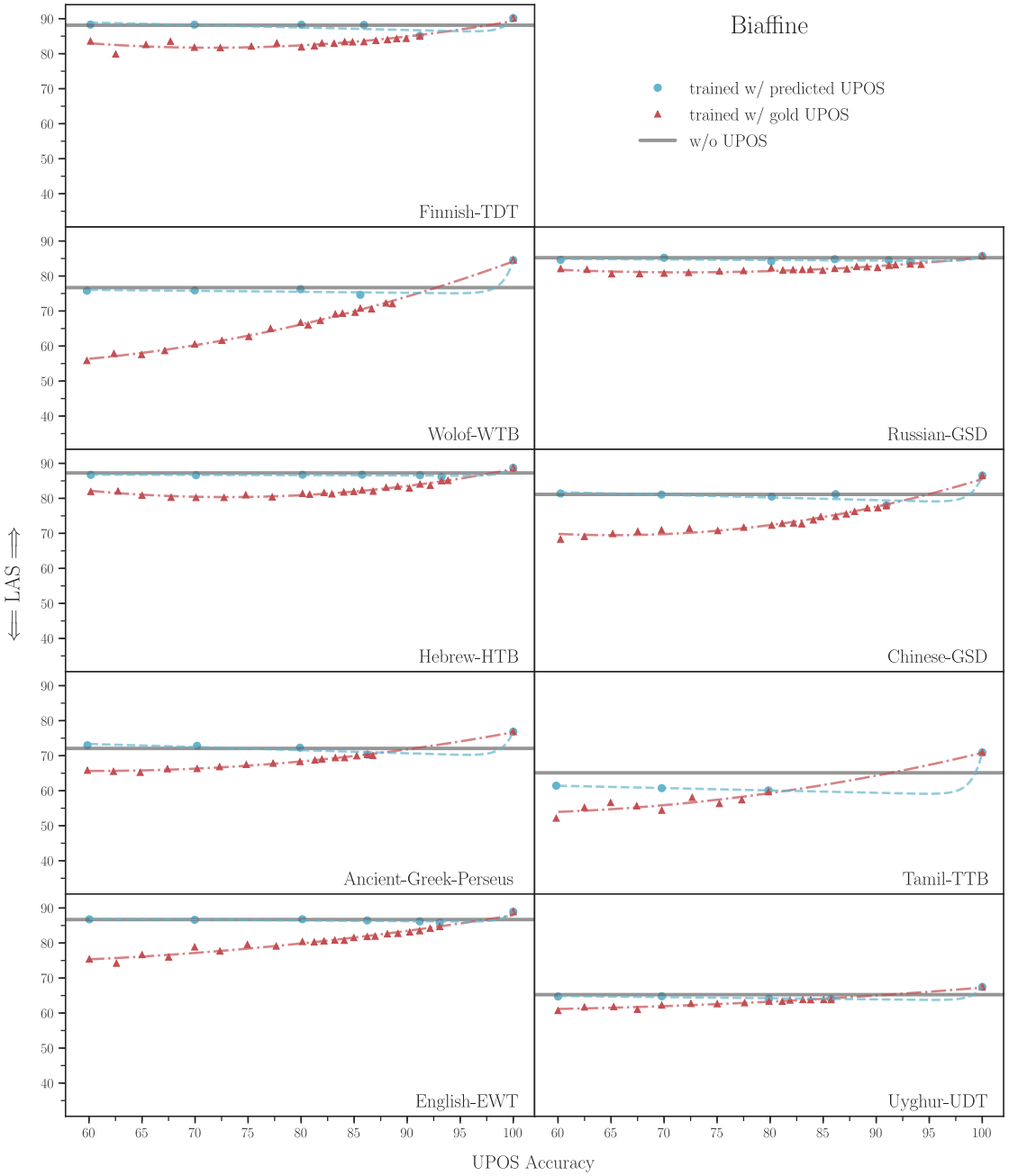


Figure 6.16: LAS for each treebank for Biaffine training with gold (red, triangles) and predicted (blue, circles) POS tags. Baseline parser trained without POS tags is shown in grey.

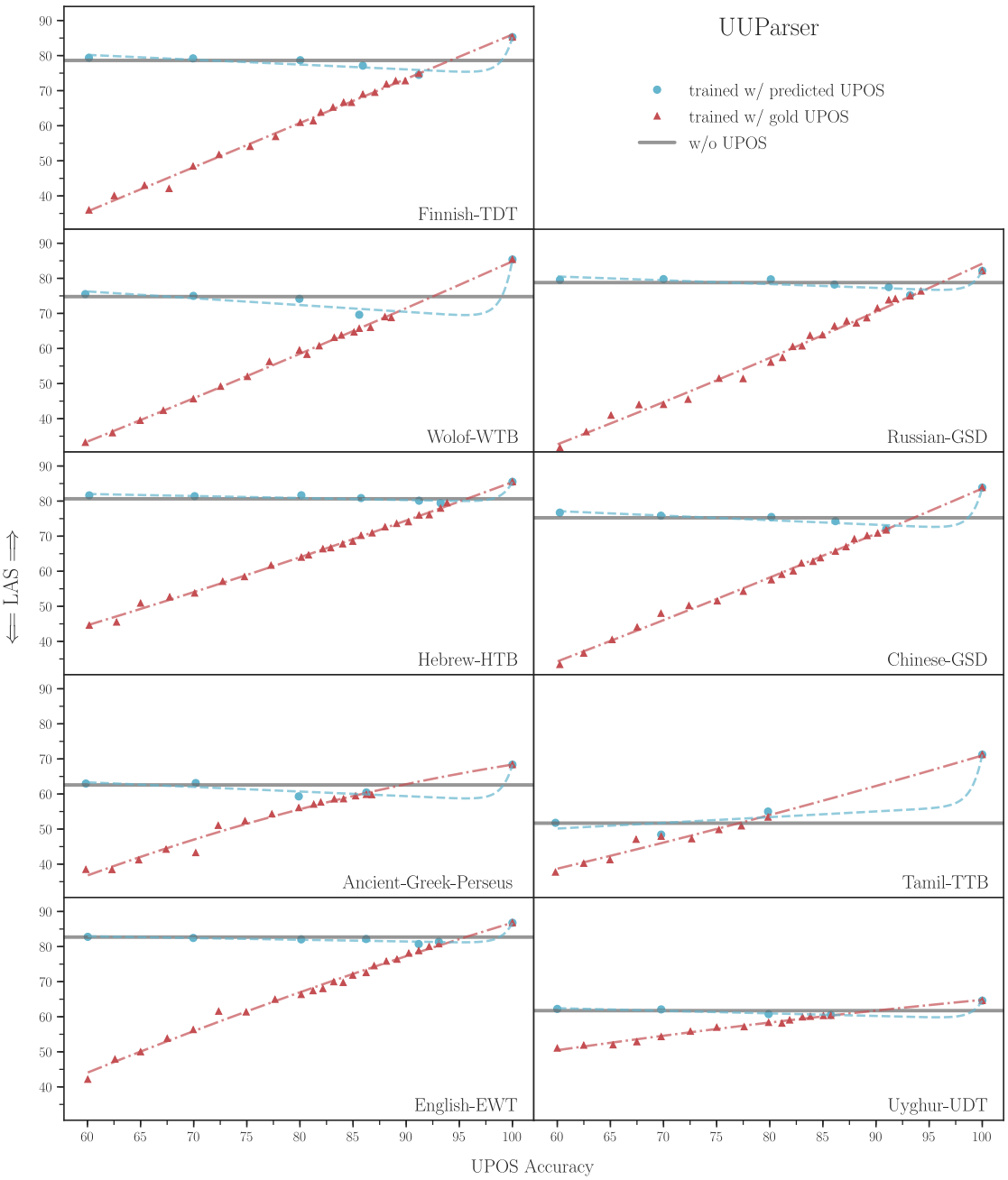


Figure 6.17: LAS for each treebank for UUParser training with gold (red, triangles) and predicted (blue, circles) POS tags. Baseline parser trained without POS tags is shown in grey.

6.A.2 Treebank performances for experiment 2 in Section 6.3

Figure 6.18: ΔUAS for each treebank for Biaffine compared to the baseline parsers trained without POS tags for different character embedding sizes and different POS tag accuracies.

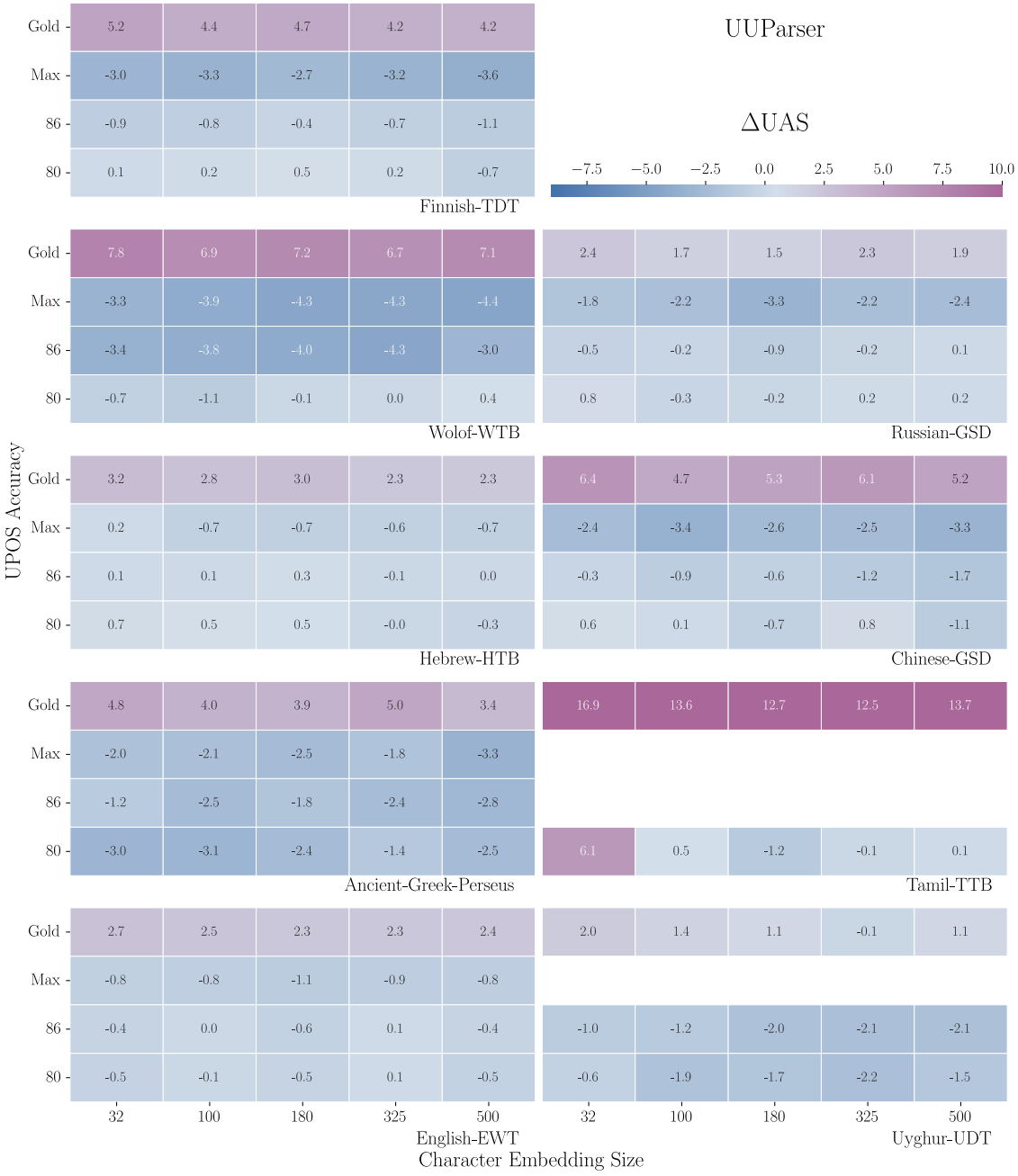


Figure 6.19: ΔUAS for each treebank for UUParser compared to the baseline parsers trained without POS tags for different character embedding sizes and different POS tag accuracies.

Figure 6.20: ΔLAS for each treebank for Biaffine compared to the baseline parsers trained without POS tags for different character embedding sizes and different POS tag accuracies.

Figure 6.21: ΔLAS for each treebank for UUParser compared to the baseline parsers trained without POS tags for different character embedding sizes and different POS tag accuracies.

6.A.3 Expanded visualisation for error analysis in Section 6.3.4

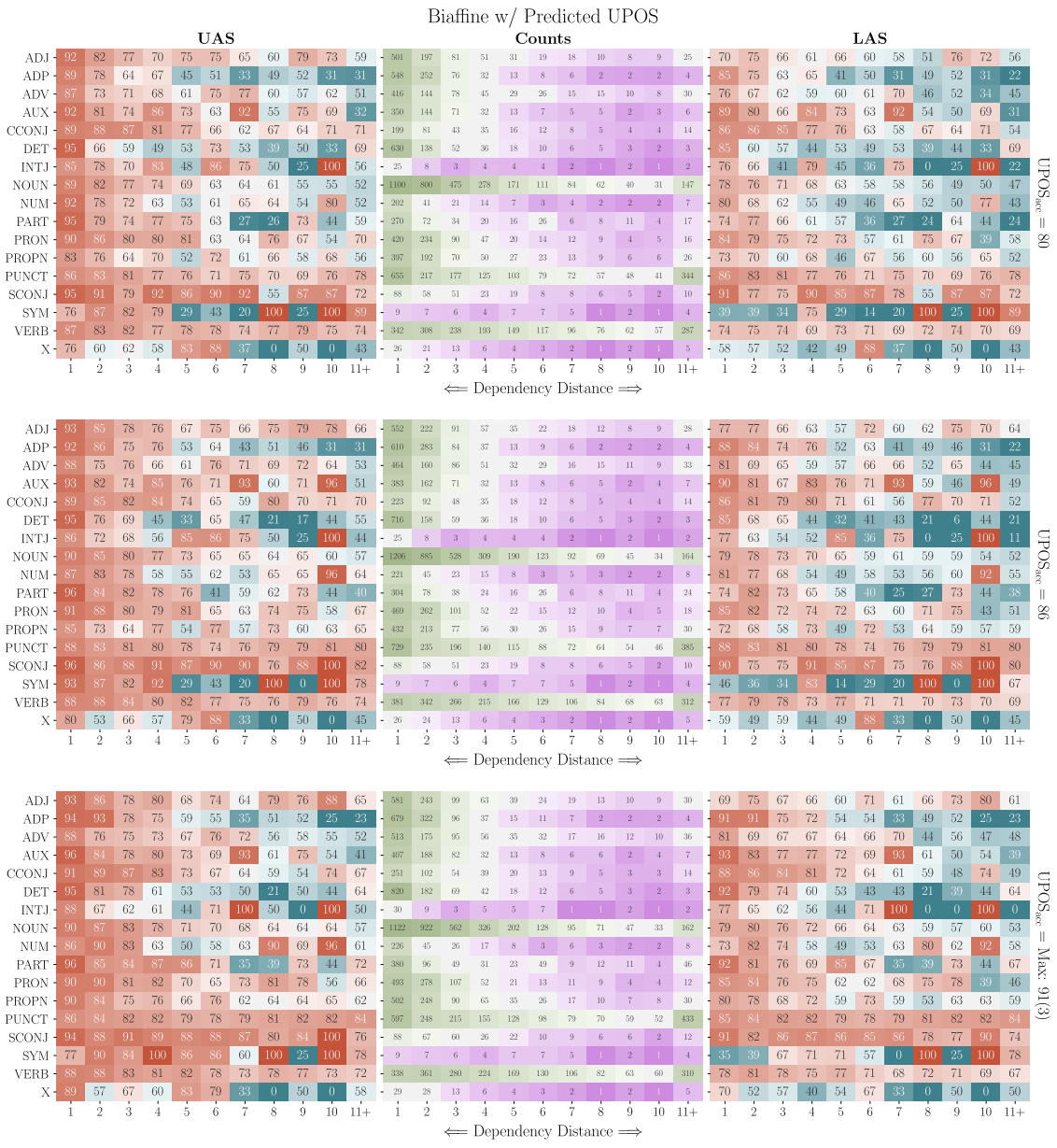


Figure 6.22: Average UAS (left column) and LAS (right column) across treebanks with Biaffine for models trained with predicted tags from taggers with 80, 86, and max POS accuracy of 91(3). UAS and LAS metrics are shown for each gold tag (y-axis) with a given dependency distance (x-axis). The counts of each pair of POS tag and dependency distance are shown in the middle column.

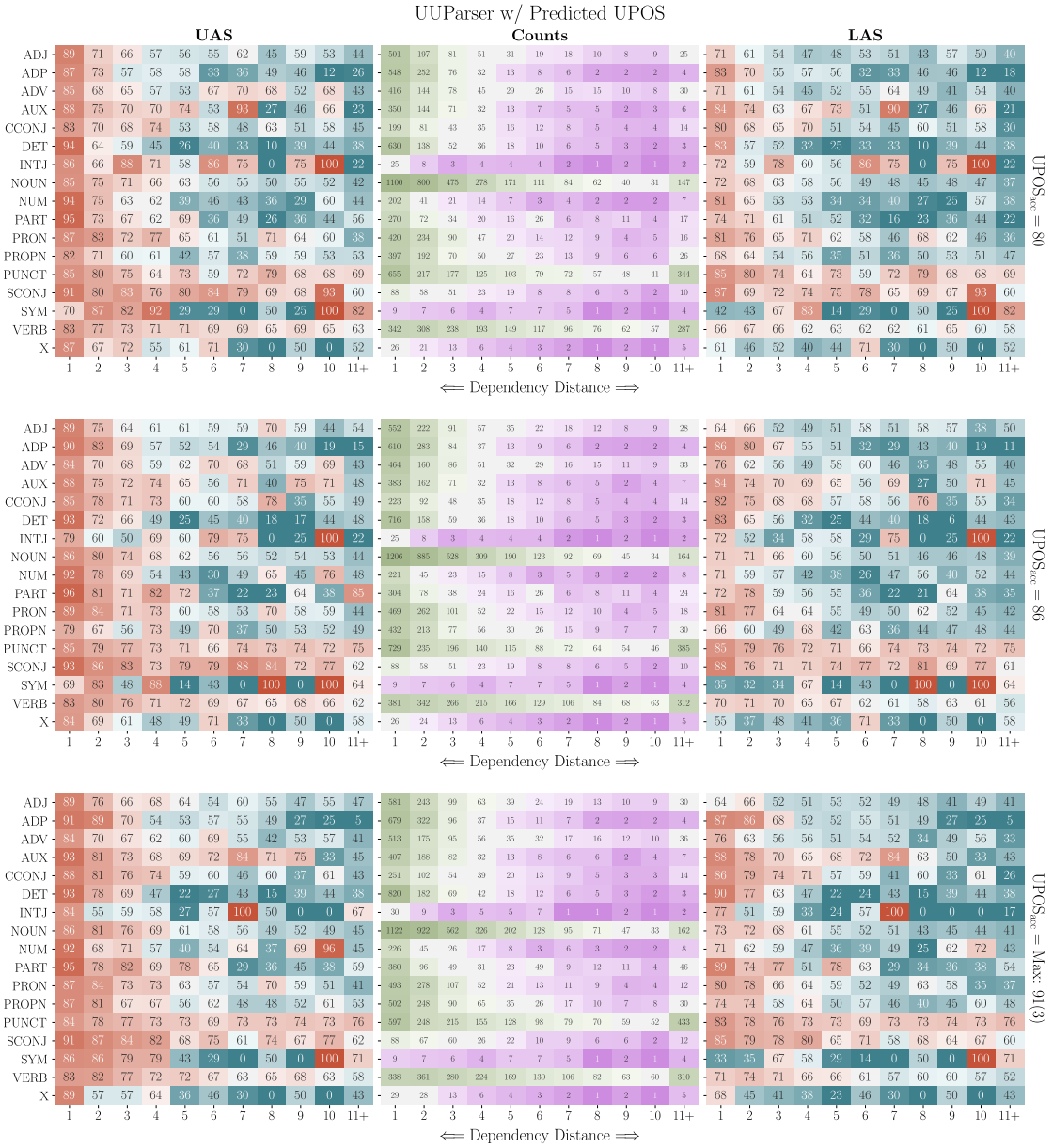


Figure 6.23: Average UAS (left column) and LAS (right column) across treebanks with UUParser for models trained with predicted tags from taggers with 80, 86, and max POS accuracy of 91(3). UAS and LAS metrics are shown for each gold tag (y-axis) with a given dependency distance (x-axis). The counts of each pair of POS tag and dependency distance are shown in the middle column.

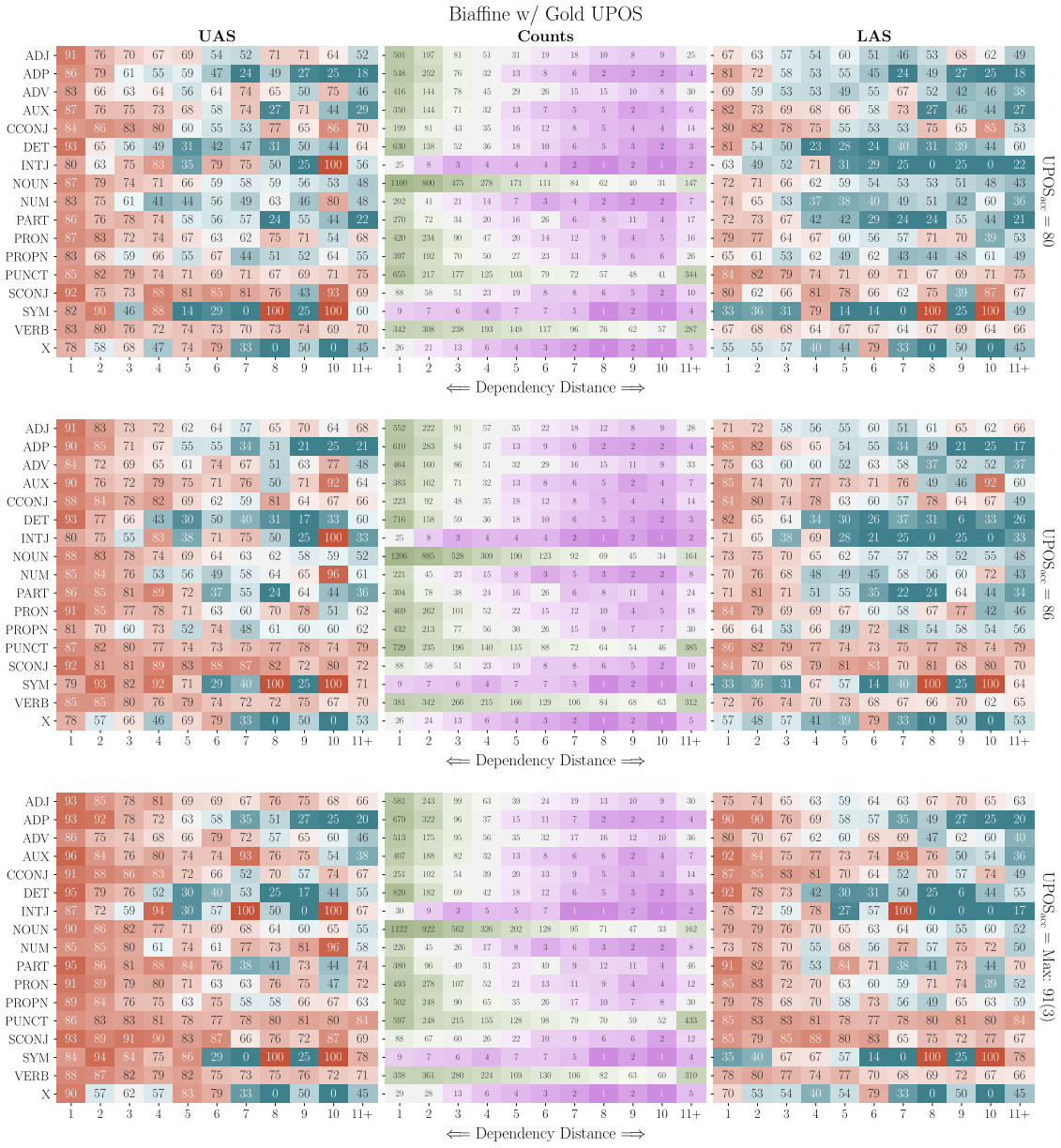


Figure 6.24: Average UAS (left column) and LAS (right column) across treebanks with Biaffine for models trained with gold tags but using predicted tags from taggers with 80, 86, and max POS accuracy of 91(3). UAS and LAS metrics are shown for each gold tag (y-axis) with a given dependency distance (x-axis). The counts of each pair of POS tag and dependency distance are shown in the middle column.

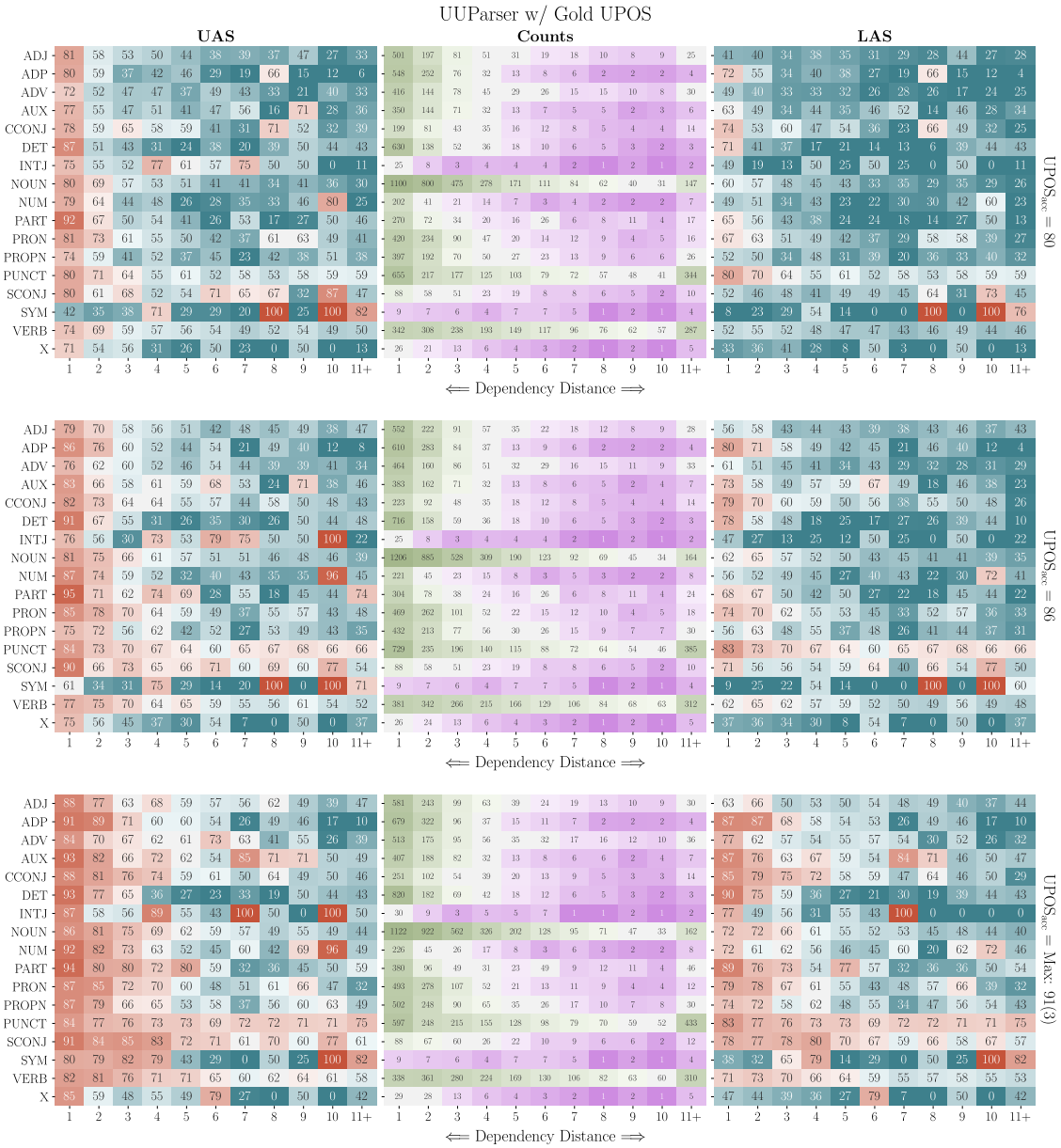


Figure 6.25: Average UAS (left column) and LAS (right column) across treebanks with UUParser for models trained with gold tags but using predicted tags from taggers with 80, 86, and max POS accuracy of 91(3). UAS and LAS metrics are shown for each gold tag (y-axis) with a given dependency distance (x-axis). The counts of each pair of POS tag and dependency distance are shown in the middle column.

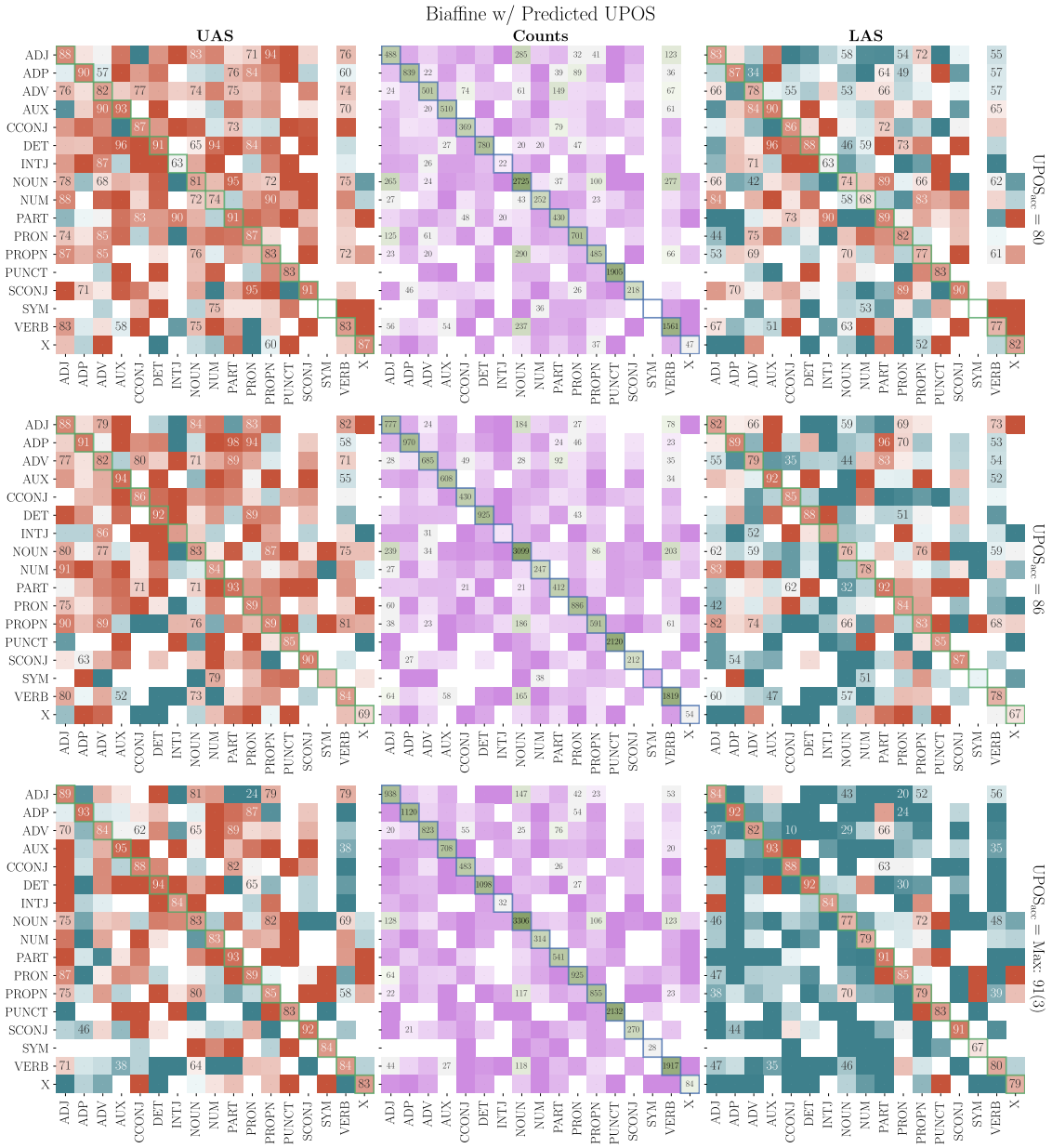


Figure 6.26: Average UAS (left column) and LAS (right column) across treebanks with Biaffine for models trained with POS tags from taggers of 80, 86, and max POS accuracy of 91(3). UAS and LAS metrics are shown for each gold tag (y-axis) predicted as any other tag (x-axis). The numbers are annotated when the average count (shown in the centre column) of a particular error is greater than 20.

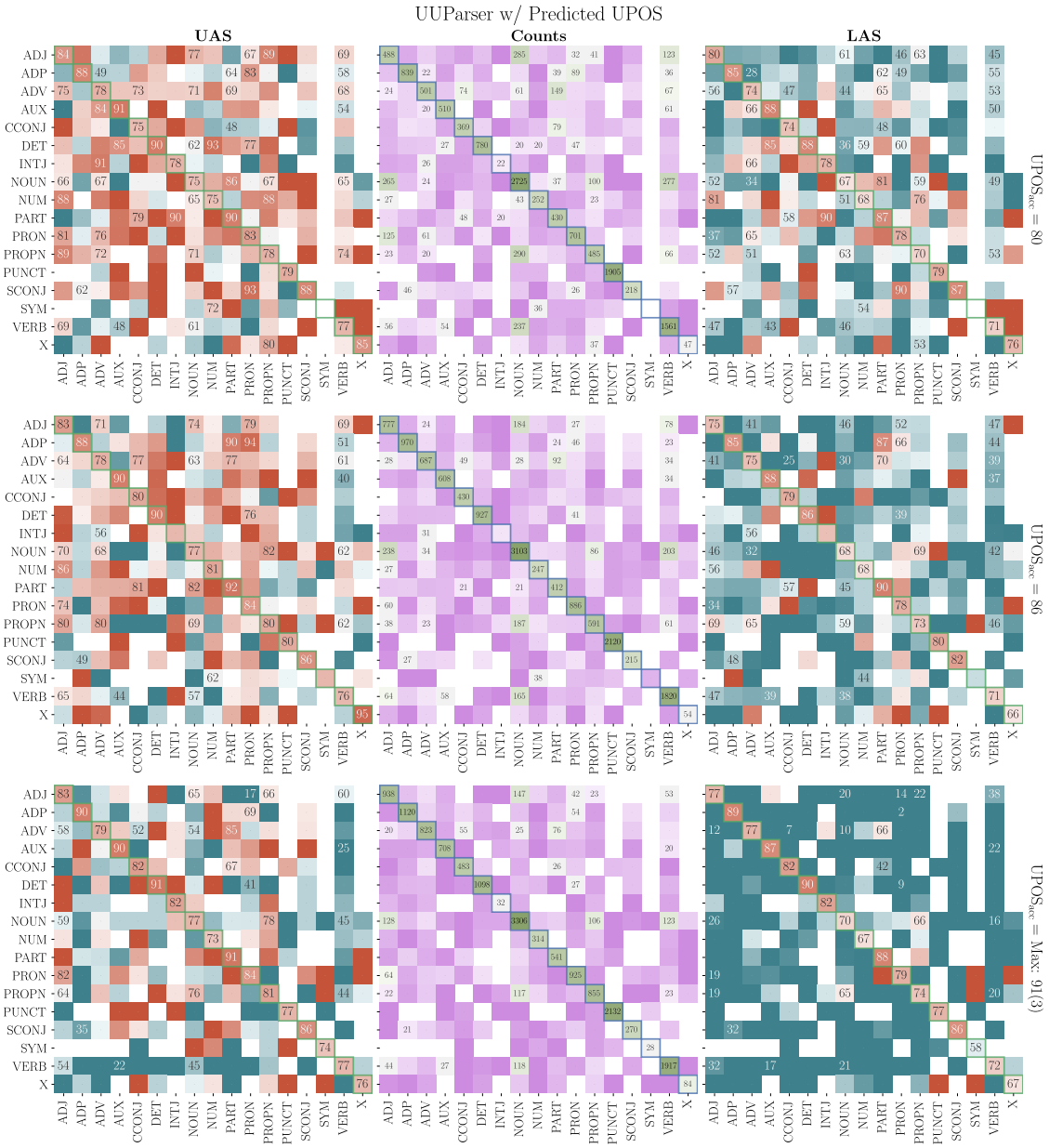


Figure 6.27: Average UAS (left column) and LAS (right column) across treebanks with UUParser for models trained with POS tags from taggers of 80, 86, and max POS accuracy of 91(3). UAS and LAS metrics are shown for each gold tag (y-axis) predicted as any other tag (x-axis). The numbers are annotated when the average count (shown in the centre column) of a particular error is greater than 20.

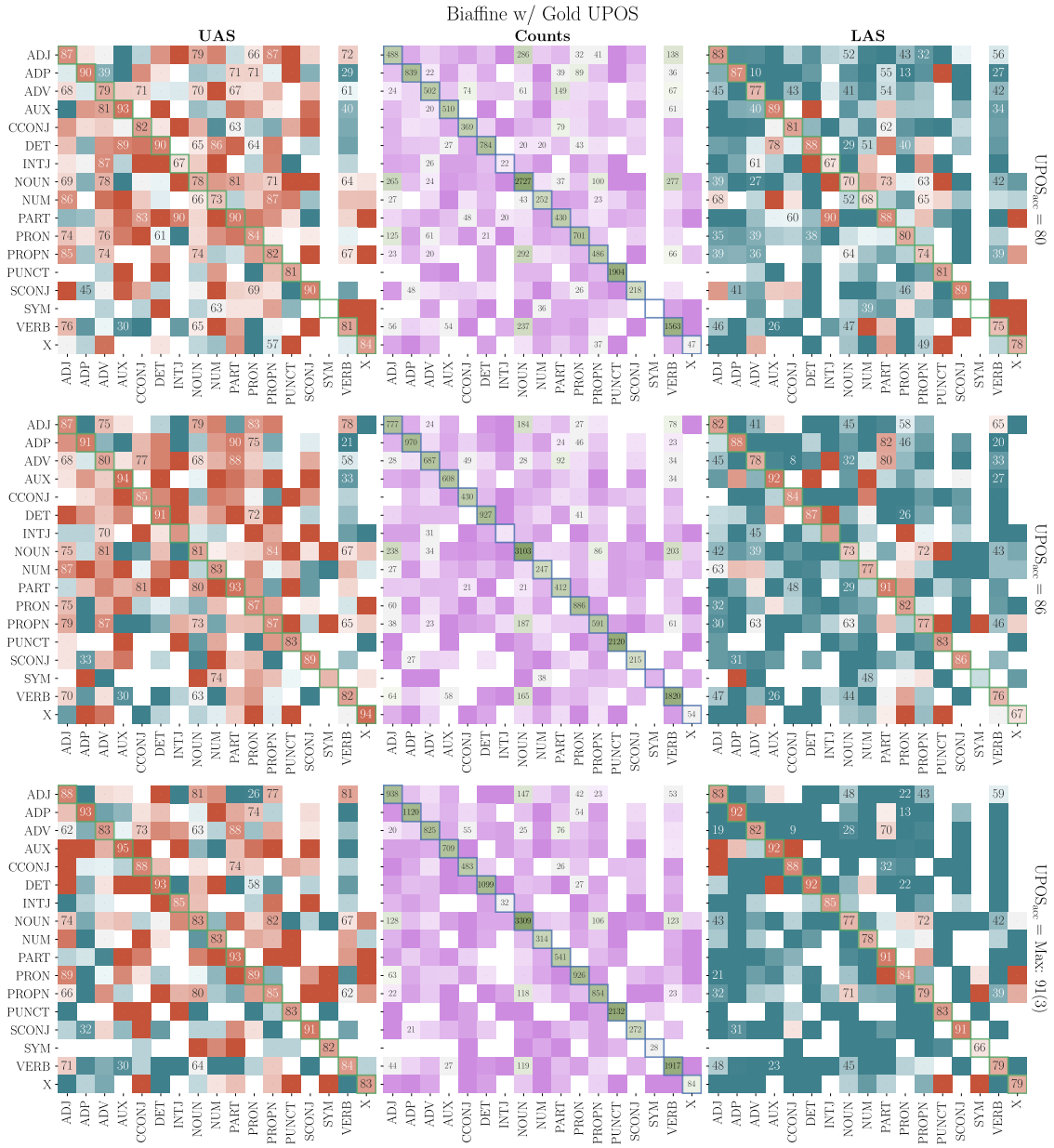


Figure 6.28: Average UAS (left column) and LAS (right column) across treebanks with Biaffine for models trained with gold tags but using predicted tags from taggers with 80, 86, and max POS accuracy of 91(3). UAS and LAS metrics are shown for each gold tag (y-axis) predicted as any other tag (x-axis). The numbers are annotated when the average count (shown in the centre column) of a particular error is greater than 20.

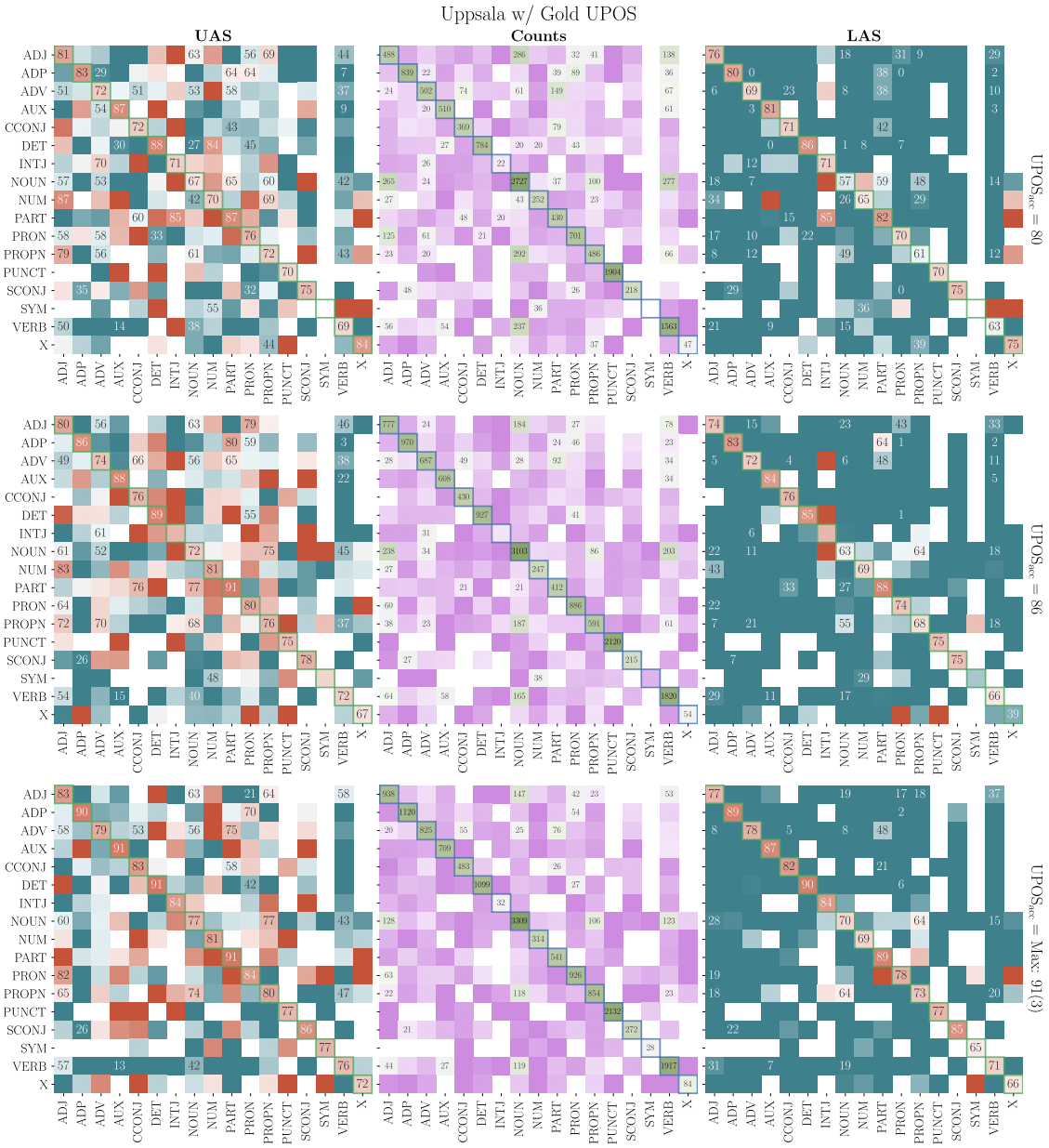


Figure 6.29: Average UAS (left column) and LAS (right column) across treebanks with UUParser for models trained with gold tags but using predicted tags from taggers with 80, 86, and max POS accuracy of 91(3). UAS and LAS metrics are shown for each gold tag (y-axis) predicted as any other tag (x-axis). The numbers are annotated when the average count (shown in the centre column) of a particular error is greater than 20.

Conclusion

First of all, we had a mixed bag of results in our endeavours to develop fast and accurate parsers. The Chunk-and-Pass technique was atrocious which in hindsight might not be so surprising. Perhaps with a much better system for modelling the composition of chunks the results would be better. But this would likely introduce more overhead and therefore slow the system down. Which is the fundamental issue with this technique. Any method introduced to optimise one of the metrics of interest severely and negatively impacts the other. But even though it failed to produce a fast and accurate parser, the work undertaken to extract chunks from treebanks has the potential to be useful. We have shown that they can moderately increase tagging and parsing accuracy, especially in a MTL setup. So some good came of it.

The distillation technique has conflicting results, however, the initial experiment was undertaken in slightly different circumstances. It appears that POS tags are actually useful in this context, so as to help the larger model distil its knowledge to the smaller network. Figure 7 shows the original Pareto front for parsers we ran locally on our machine in a consistent setting with the two fastest distilled models from Chapter 4. These models used the standard predicted POS tags as features and clearly push the boundary of the Pareto front, especially the smaller model on CPU.

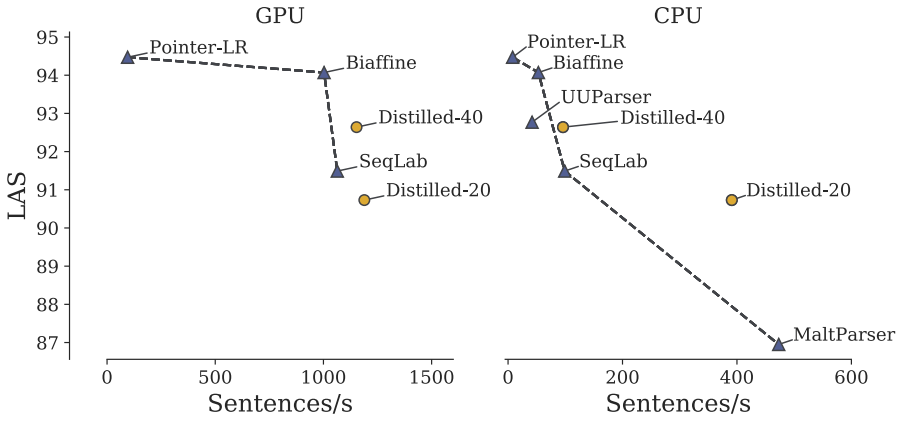


Figure 7: Pareto front of modern parsers run on our machine locally with parsers developed during thesis. Previous models in blue triangles and distilled models (Chapter 4) in yellow circles.

The work in Chapter 5 offered some explanation towards differences observed in parsing with respect to different algorithms and also differences in data. The work focusing on transition-based algorithms shed some light on a long-standing puzzle. It showed that a large proportion of the variation observed for different algorithms for different treebanks can be explained by how similar the distribution of dependency displacement in a treebank is to the inherent distribution an algorithm is biased towards. This required undertaking a sentence-length binning analysis which is perhaps a clear general takeaway from this thesis. That analyses focusing on or using linguistic features that are related to sentence lengths

almost always benefit from a sentence-length binning procedure. Otherwise spurious results abound or meaningful phenomena are hidden. The second part of this chapter then focused on the similarity of these distributions between the training and test data of treebanks. We observed a strong correlation even when controlling for covariants and discussed a practical use of this finding. We gave a working demonstration of using this to guide an adversarial sampling procedure to more thoroughly evaluate parsers.

Finally, we investigated how useful POS tags are for parsers in a number of different contexts. In the main analysis, we obtained results suggesting that even predicted POS tags from very accurate taggers were worse than not using any at all and that gold-standard POS tags have some exceptionality. We also observed that the smallest treebank did benefit from predicted tags even though the accuracy was lower than for most treebanks. We used these two findings (gold tag exceptionality and smaller treebanks benefiting from tags) to develop two secondary analyses. We evaluated why gold tags are so useful by establishing what parsers inherently learn about POS tags. We found that parsers seem to learn a substantial amount about POS tags and what they fail to learn has a strong crossover to what taggers also fail to learn. In a number of masking experiments, we observed that it was indeed the situations where the taggers failed to predict the POS tags accurately that they were most useful. Then we expanded the work on smaller treebanks and investigated the utility of POS tags in low-resource settings. We consolidated the findings of the original analysis using artificial low-resource data, real low- and lower-resource data, and also low-resource data with augmented data. It was clear that even when the predicted POS tags came from taggers only achieving a low accuracy, for low-resource parsing they are still useful. Fundamentally, the work on the efficacy of POS tags merely highlights the need to evaluate whether they are beneficial for a given language in a given context rather than taking it for granted that they will in the worst case add nothing when in fact they can actually harm performance.

References

S. Abney. 1997. Part-of-speech tagging and partial parsing. In Steve Young and Gerrit Bloothooft, editors, *Corpus-Based Methods in Language and Speech Processing*, pages 118–136. Springer Netherlands, Dordrecht.

Željko Agić, Anders Johannsen, Barbara Plank, Héctor Martínez Alonso, Natalie Schluter, and Anders Søgaard. 2016. Multilingual projection for parsing truly low-resource languages. *Transactions of the Association for Computational Linguistics*, 4:301–312.

Ramadan Alfared and Denis Béchet. 2012. POS taggers and dependency parsing. *International Journal of Computational Linguistics and Applications*, 3(2):107–122.

Anita Alicante, Cristina Bosco, Anna Corazza, and Alberto Lavelli. 2012. A treebank-based study on the influence of italian word order on parsing performance. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC-2012)*, pages 1985–1992.

American Psychological Association. 2010. *Publication manual of the American Psychological Association*, 6th edition. American Psychological Association, Washington, DC.

Waleed Ammar, George Mulcaire, Miguel Ballesteros, Chris Dyer, and Noah A Smith. 2016. Many languages, one parser. *Transactions of the Association for Computational Linguistics*, 4:431–444.

Mark Anderson and Carlos Gómez-Rodríguez. 2020a. Distilling neural networks for greener and faster dependency parsing. In *Proceedings of the 16th International Conference on Parsing Technologies and the IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies*, pages 2–13, Online. Association for Computational Linguistics.

Mark Anderson and Carlos Gómez-Rodríguez. 2020b. Inherent dependency displacement bias of transition-based algorithms. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 5147–5155, Marseille, France. European Language Resources Association.

Mark Anderson and Carlos Gómez-Rodríguez. 2020c. On the frailty of universal POS tags for neural UD parsers. In *Proceedings of the 24th Conference on Computational Natural Language Learning*, pages 69–96, Online. Association for Computational Linguistics.

Mark Anderson and Carlos Gómez-Rodríguez. 2021. What taggers fail to learn, parsers need the most. In *Proceedings of the 23rd Nordic Conference of Computational Linguistics (NoDaLiDa 2021)*, Online.

Mark Anderson, Carlos Gómez Rodríguez, and Anders Søgaard. 2021. Replicating and extending "Because their treebanks leak": Graph isomorphism, covariants, and parser performance. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics*, Online. Association for Computational Linguistics.

Mark Anderson, David Vilares, and Carlos Gómez-Rodríguez. 2019. Artificially evolved chunks for morphosyntactic analysis. In *Proceedings of the 18th International Workshop on Treebanks and Linguistic Theories (TLT, SyntaxFest 2019)*, pages 133–143, Paris, France. Association for Computational Linguistics.

Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2017. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3):32.

Gluseppe Attardi and Felice Dell'Orletta. 2008. Chunking and dependency parsing. In *Proceedings of LREC Workshop on Partial Parsing: Between Chunking and Deep Parsing.*

Jimmy Ba and Rich Caruana. 2014. Do deep nets really need to be deep? In *Advances in Neural Information Processing Systems*, pages 2654–2662.

Thomas Back. 1996. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford university press.

Jiangang Bai, Yujing Wang, Yiren Chen, Yaming Yang, Jing Bai, Jing Yu, and Yunhai Tong. 2021. Syntax-BERT: Improving pre-trained transformers with syntax trees. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 3011–3020, Online. Association for Computational Linguistics.

Jason Baldridge and Alexis Palmer. 2009. How well does active learning actually work? time-based evaluation of cost-reduction strategies for language documentation. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 296–305.

Miguel Ballesteros, Chris Dyer, and Noah A. Smith. 2015. Improved transition-based parsing by modeling characters instead of words with LSTMs. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 349–359, Lisbon, Portugal. Association for Computational Linguistics.

Miguel Ballesteros and Joakim Nivre. 2012. MaltOptimizer: an optimization tool for MaltParser. In *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 58–62. Association for Computational Linguistics.

Miguel Ballesteros and Joakim Nivre. 2013. Going to the roots of dependency parsing. *Computational Linguistics*, 39(1):5–13.

Aleksandrs Berdicevskis, Çağrı Çöltekin, Katharina Ehret, Kilu von Prince, Daniel Ross, Bill Thompson, Chunxiao Yan, Vera Demberg, Gary Lupyan, Taraka Rama, and Christian Bentz. 2018. Using Universal Dependencies in cross-linguistic complexity research. In *Second Workshop on Universal Dependencies,November 1, 2018, Brussels, Belgium*, pages 8–17.

Riyaz Ahmad Bhat, Rajesh Bhatt, Annahita Farudi, Prescott Klassen, Bhuvana Narasimhan, Martha Palmer, Owen Rambow, Dipti Misra Sharma, Ashwini Vaidya, Sri Ramagurumurthy Vishnu, et al. 2017. The hindi/urdu treebank project. In *Handbook of Linguistic Annotation*, pages 659–697. Springer.

Dana Bielec. 1998. *Polish: An Essential Grammar*. Routledge, London.

Bernd Bohnet and Joakim Nivre. 2012. A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1455–1465, Jeju Island, Korea. Association for Computational Linguistics.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146.

C. Bosco, S. Montemagni, A. Mazzei, V. Lombardo, F. Dell'orletta, Alessandro Lenci, L. Lesmo, Giuseppe Attardi, Maria Simi, A. Lavelli, J. Hall, J. Nilsson, and J. Nivre. 2010. Comparing the influence of different treebank annotations on dependency parsing performance. In *7th international conference on Language Resources and Evaluation (LREC 2010)*, volume 1, pages 1794–1801.

G. Bouma. 2009. Normalized (pointwise) mutual information in collocation extraction. *From Form to Meaning: Processing Texts Automatically, Proceedings of the Biennial GSCL Conference 2009*.

Gosse Bouma, Djamé Seddah, and Daniel Zeman. 2020. Overview of the IWPT 2020 shared task on parsing into enhanced Universal Dependencies. In *Proceedings of the 16th International Conference on Parsing Technologies and the IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies*, pages 151–161, Online. Association for Computational Linguistics.

Lucien Brown. 2015. Honorifics and politeness. *The handbook of Korean linguistics*, pages 303–319.

Matthias Buch-Kromann. 2006. *Discontinuous grammar: A dependency-based model of human parsing and language learning*. Ph.D. thesis, Copenhagen Business School.

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pages 149–164, New York City. Association for Computational Linguistics.

Cristian Bucilă, Rich Caruana, and Alexandru Niculescu-Mizil. 2006. Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 535–541. ACM.

Xavier Carreras, Mihai Surdeanu, and Lluís Màrquez. 2006. Projective dependency parsing with perceptron. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pages 181–185, New York City. Association for Computational Linguistics.

Suk-Jin Chang. 1996. *Korean*. John Benjamins, Philadelphia.

Wanxiang Che, Yijia Liu, Yuxuan Wang, Bo Zheng, and Ting Liu. 2018. Towards better UD parsing: Deep contextualized word embeddings, ensemble, and treebank concatenation. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 55–64, Brussels, Belgium. Association for Computational Linguistics.

Danqi Chen and Christopher Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 740–750.

Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2018. Model compression and acceleration for deep neural networks: The principles, progress, and challenges. *IEEE Signal Processing Magazine*, 35(1):126–136.

Luis Chiruzzo and Dina Wonsever. 2020. Statistical deep parsing for Spanish using neural networks. In *Proceedings of the 16th International Conference on Parsing Technologies and the IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies*, pages 132–144, Online. Association for Computational Linguistics.

Key-Sun Choi, Young S Han, Young G Han, and Oh W Kwon. 1994. Kaist tree bank project for korean: Present and future development. In *Proceedings of the International Workshop on Sharable Natural Language Resources*, pages 7–14. Citeseer.

Noam Chomsky. 1957. *Syntactic Structures*. Mouton and Co., The Hague.

Shammur Absar Chowdhury and Roberto Zamparelli. 2018. Rnn simulations of grammaticality judgments on long-distance dependencies. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 133–144.

Morten H. Christiansen and Nick Chater. 2016. The now-or-never bottleneck: A fundamental constraint on language. *Behavioral and Brain Sciences*, 39.

Yoeng-Jin Chu and Tseng-Hong Liu. 1965. On the shortest arborescence of a directed graph. *Scientia Sinica*, 14:1396–1400.

Jayeol Chun, Na-Rae Han, Jena D Hwang, and Jinho D Choi. 2018. Building Universal Dependency treebanks in korean. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*.

Tagyoung Chung, Matt Post, and Daniel Gildea. 2010. Factors affecting the accuracy of korean parsing. In *Proceedings of the NAACL HLT 2010 First Workshop on Statistical Parsing of Morphologically-Rich Languages*, pages 49–57.

Fabio Ciravegna and Alberto Lavelli. 1999. Full text parsing using cascades of rules: an information extraction perspective. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*.

Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. 2019a. What does BERT look at? An analysis of BERT's attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, Florence, Italy. Association for Computational Linguistics.

Kevin Clark, Minh-Thang Luong, Urvashi Khandelwal, Christopher D Manning, and Quoc Le. 2019b. BAM! Born-again multi-task networks for natural language understanding. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5931–5937.

Kevin Clark, Minh-Thang Luong, Christopher D. Manning, and Quoc V. Le. 2018. Semi-supervised sequence modeling with cross-view training. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1914–1925.

Federica Cognola and Jan Casalicchio. 2018. On the null-subject phenomenon. *Null subjects in Generative Grammar. A Synchronic and Diachronic Perspective*, pages 1–28.

Çağrı Çöltekin. 2020. Verification, reproduction and replication of NLP experiments: a case study on parsing Universal Dependencies. In *Proceedings of the Fourth Workshop on Universal Dependencies (UDW 2020)*, pages 46–56, Barcelona, Spain (Online). Association for Computational Linguistics.

Bernard Comrie. 1978. Ergativity. In Winfred P. Lehmann, editor, *Syntactic Typology: Studies in the Phenomenology of Language*, pages 329–394. University of Texas Press, Austin.

Anna Corazza, Alberto Lavelli, and Giorgio Satta. 2013. An information-theoretic measure to evaluate parsing difficulty across treebanks. *ACM Trans. Speech Lang. Process.*, 9(4).

Michael A Covington. 1990. A dependency parser for variable-word-order languages. *Technical ReportAI-1990-01*.

Michael A Covington. 2001. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th annual ACM southeast conference*, volume 1. Citeseer.

Mary Dalrymple. 2006. How much can part-of-speech tagging help parsing? *Natural Language Engineering*, 12(4):373–389.

Dipanjan Das and Slav Petrov. 2011. Unsupervised part-of-speech tagging with bilingual graph-based projections. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 600–609.

Mathieu Dehouck, Mark Anderson, and Carlos Gómez-Rodríguez. 2020. Efficient EUD parsing. In *Proceedings of the 16th International Conference on Parsing Technologies and the IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies*, pages 192–205, Online. Association for Computational Linguistics.

Mathieu Dehouck and Pascal Denis. 2018. A framework for understanding the role of morphology in universal dependency parsing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2864–2870.

Mathieu Dehouck and Carlos Gómez-Rodríguez. 2020. Data augmentation via subtree swapping for dependency parsing of low-resource languages. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 3818–3830.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186.

Cheikh M Bamba Dione. 2019. Developing Universal Uependencies for Wolof. In *Proceedings of the Third Workshop on Universal Dependencies (UDW, SyntaxFest 2019)*, pages 12–23.

Cícero Nogueira Dos Santos and Bianca Zadrozny. 2014. Learning character-level representations for part-of-speech tagging. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page II–1818–II–1826. JMLR.org.

Timothy Dozat and Christopher D. Manning. 2017. Deep biaffine attention for neural dependency parsing. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Timothy Dozat and Christopher D. Manning. 2018. Simpler but more accurate semantic dependency parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 484–490, Melbourne, Australia. Association for Computational Linguistics.

Timothy Dozat, Peng Qi, and Christopher D. Manning. 2017. Stanford's graph-based neural dependency parser at the CoNLL 2017 shared task. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 20–30, Vancouver, Canada. Association for Computational Linguistics.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 334–343, Beijing, China. Association for Computational Linguistics.

Jack Edmonds. 1967. Optimum branchings. *Journal of Research of the national Bureau of Standards B*, 71(4):233–240.

Jason Eisner. 2000. Bilexical grammars and their cubic-time parsing algorithms. In *Advances in probabilistic and other parsing technologies*, pages 29–61. Springer.

Jason Eisner and Noah A Smith. 2010. Favor short dependencies: Parsing with soft and hard constraints on dependency length. In *Trends in Parsing Technology*, pages 121–150. Springer.

Jason M. Eisner. 1996. Three new probabilistic models for dependency parsing: an exploration. In *COLING '96 Proceedings of the 16th conference on Computational linguistics - Volume 1*, pages 340–345.

Agnieszka Falenska, Anders Björkelund, and Jonas Kuhn. 2020. Integrating graph-based and transition-based dependency parsers in the deep contextualized era. In *Proceedings of the 16th International Conference on Parsing Technologies and the IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies*, pages 25–39, Online. Association for Computational Linguistics.

Agnieszka Falenska and Özlem Çetinoğlu. 2017. Lexicalized vs. delexicalized parsing in low-resource scenarios. In *Proceedings of the 15th International Conference on Parsing Technologies*, pages 18–24, Pisa, Italy. Association for Computational Linguistics.

Murhaf Fares, Stephan Oepen, Lilja Øvrelid, Jari Björne, and Richard Johansson. 2018. The 2018 shared task on extrinsic parser evaluation: On the downstream utility of English Universal Dependency parsers. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 22–33, Brussels, Belgium. Association for Computational Linguistics.

Donald E Farrar and Robert R Glauber. 1967. Multicollinearity in regression analysis: the problem revisited. *The Review of Economic and Statistics*, pages 92–107.

Ronald F Feldstein. 2001. *A concise Polish grammar*. Citeseer.

Daniel Fernández-González and Carlos Gómez-Rodríguez. 2018. Non-projective dependency parsing with non-local transitions. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 693–700, New Orleans, Louisiana. Association for Computational Linguistics.

Daniel Fernández-González and Carlos Gómez-Rodríguez. 2019. Left-to-right dependency parsing with pointer networks. In *Proceedings of NAACL-HLT*, pages 710–716.

Daniel Fernández-González, Carlos Gómez-Rodríguez, and David Vilares. 2016. Improving the arc-eager model with reverse parsing. *Computing and Informatics*, 35(3):555–585.

Ramon Ferrer-i-Cancho. 2004. Euclidean distance between syntactically linked words. *Physical Review E*, 70(5):056135.

Ramon Ferrer-i-Cancho and Carlos Gómez-Rodríguez. 2016. Crossings as a side effect of dependency lengths. *Complexity*, 21(S2):320–328.

Ramon Ferrer-i-Cancho and Haitao Liu. 2014. The risks of mixing dependency lengths from sequences of different length. *Glottotheory*, 5(2):143–155.

R. de Fleury. 1934. *1700 Cocktails for the Man behind the Bar*. William Heinemann Ltd.

Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research*, 13:2171–2175.

Jennifer Foster. 2010. "cba to check the spelling": Investigating parser performance on discussion forum posts. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 381–384.

Richard Futrell, Kyle Mahowald, and Edward Gibson. 2015. Large-scale evidence of dependency length minimization in 37 languages. *Proceedings of the National Academy of Sciences*, 112(33):10336–10341.

Richard Futrell, Ethan Wilcox, Takashi Morita, and Roger Levy. 2018. RNNs as psycholinguistic subjects: Syntactic state and grammatical dependency. *arXiv preprint arXiv:1809.01329*.

Richard Futrell, Ethan Wilcox, Takashi Morita, Peng Qian, Miguel Ballesteros, and Roger Levy. 2019. Neural language models as psycholinguistic subjects: Representations of syntactic state. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 32–42.

Haim Gaifman. 1965. Dependency systems and phrase-structure systems. *Information & Computation*, 8(3):304–337.

Pablo Gamallo. 2015. Dependency parsing with compression rules. In *Proceedings of the 14th International Conference on Parsing Technologies*, pages 107–117.

Kuzman Ganchev, Jennifer Gillenwater, and Ben Taskar. 2009. Dependency grammar induction via bitext projection constraints. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 369–377.

Dan Garrette, Jason Mielens, and Jason Baldridge. 2013. Real-world semi-supervised learning of pos-taggers for low-resource languages. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 583–592.

Kim Gerdes, Bruno Guillaume, Sylvain Kahane, and Guy Perrier. 2018. SUD or surface-syntactic Universal Dependencies: An annotation scheme near-isomorphic to UD. In *Proceedings of the Second Workshop on Universal Dependencies (UDW 2018)*, pages 66–74, Brussels, Belgium. Association for Computational Linguistics.

Edward Gibson. 2000. The dependency locality theory: A distance-based theory of linguistic complexity. *Image, language, brain*, 2000:95–126.

Daniel Gildea. 2001. Corpus variation and parser performance. In *Proceedings of the 2001 Conference on Empirical Methods in Natural Language Processing*.

Daniel Gildea and David Temperley. 2010. Do grammars minimize dependency length? *Cognitive Science*, 34(2):286–310.

Filip Ginter, Jan Hajic, Juhani Luotolahti, Milan Straka, and Daniel Zeman. 2017. CoNLL 2017 shared task-automatically annotated raw texts and word embeddings. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University.

Mario Giulianelli, Jack Harding, Florian Mohnert, Dieuwke Hupkes, and Willem Zuidema. 2018. Under the hood: Using diagnostic classifiers to investigate and improve how language models track agreement information. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 240–248, Brussels, Belgium. Association for Computational Linguistics.

Goran Glavaš and Ivan Vulić. 2021. Is supervised syntactic parsing beneficial for language understanding tasks? an empirical investigation. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 3090–3104, Online. Association for Computational Linguistics.

Yoav Goldberg. 2019. Assessing BERT's syntactic abilities. *arXiv preprint arXiv:1901.05287*.

Carlos Gómez-Rodríguez. 2017. On the relation between dependency distance, crossing dependencies, and parsing. *Physics of Life Reviews*, 21:200 – 203.

Carlos Gómez-Rodríguez, Iago Alonso-Alonso, and David Vilares. 2019. How important is syntactic parsing accuracy? an empirical evaluation on rule-based sentiment analysis. *Artificial Intelligence Review*, 52(3):2081–2097.

Carlos Gómez-Rodríguez and Daniel Fernández-González. 2015. An efficient dynamic oracle for unrestricted non-projective parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 256–261, Beijing, China. Association for Computational Linguistics.

Carlos Gómez-Rodríguez and Ramon Ferrer-i-Cancho. 2017. Scarcity of crossing dependencies: A direct outcome of a specific constraint? *Physical Review E*, 96(6):062304.

Carlos Gómez-Rodríguez, Michalina Strzyz, and David Vilares. 2020. A unifying theory of transition-based and sequence labeling parsing. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 3776–3793, Barcelona, Spain (Online). International Committee on Computational Linguistics.

Carlos Gómez-Rodríguez and David Vilares. 2018. Constituent parsing as sequence labeling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 1314–1324. Association for Computational Linguistics.

Kyle Gorman and Steven Bedrick. 2019. We need to talk about standard splits. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2786–2791, Florence, Italy. Association for Computational Linguistics.

Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. 2018. Learning word vectors for 157 languages. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*.

Nathan Green, Loganathan Ramasamy, and Zdeněk Žabokrtskỳ. 2012. Using an svm ensemble system for improved tamil dependency parsing. In *Proceedings of the ACL 2012 Joint Workshop on Statistical Parsing and Semantic Processing of Morphologically Rich Languages*, pages 72–77.

Ralph Grishman and Beth Sundheim. 1996. Message Understanding Conference- 6: A brief history. In *COLING 1996 Volume 1: The 16th International Conference on Computational Linguistics*.

Ulrike Groemping. 2006. Relative importance for linear regression in r: The package relaimpo. *Journal of Statistical Software*, 17(1):1–27.

Aleksandra Gruszka and Edward Nęcka. 2017. Limitations of working memory capacity: The cognitive and social consequences. *European Management Journal*, 35(6):776–784. Managing Overflows.

Bruno Guillaume, Marie-Catherine de Marneffe, and Guy Perrier. 2019. Conversion et améliorations de corpus du français annotés en Universal Dependencies. *Traitement Automatique des Langues*, 60(2):71–95.

Kristina Gulordava, Piotr Bojanowski, Edouard Grave, Tal Linzen, and Marco Baroni. 2018. Colorless green recurrent networks dream hierarchically. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1195–1205, New Orleans, Louisiana. Association for Computational Linguistics.

Kristina Gulordava and Paola Merlo. 2015. Diachronic trends in word order freedom and dependency length in dependency-annotated corpora of Latin and Ancient Greek. In *Proceedings of the third international conference on dependency linguistics (Depling 2015)*, pages 121–130.

Kristina Gulordava and Paola Merlo. 2016. Multi-lingual dependency parsing evaluation: a large-scale analysis of word order properties using artificial data. *Transactions of the Association for Computational Linguistics*, 4:343–356.

Masafumi Hagiwara. 1994. A simple and effective method for removal of hidden units and weights. *Neurocomputing*, 6(2):207–218.

Jan Hajič, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia Martí, Lluís Màrquez, Adam Meyers, Joakim Nivre, Sebastian Padó, Jan Štěpánek, Pavel Straňák, Mihai Surdeanu, Nianwen Xue, and Yi Zhang. 2009. The CoNLL-2009 shared task: Syntactic and semantic dependencies in multiple languages. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL 2009): Shared Task*, pages 1–18, Boulder, Colorado. Association for Computational Linguistics.

David Hall, Taylor Berg-Kirkpatrick, and Dan Klein. 2014. Sparser, better, faster GPU parsing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 208–217.

Song Han, Huizi Mao, and William J Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *ICLR*.

Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143.

Martin Haspelmath. 2011. The indeterminacy of word segmentation and the nature of morphology and syntax. *Folia Linguistica*, 45(1):31–80.

Katri Haverinen, Filip Ginter, Veronika Laippala, and Tapio Salakoski. 2009. Parsing clinical Finnish: Experiments with rule-based and statistical dependency parsers. In *Proceedings of the 17th Nordic Conference of Computational Linguistics (NODALIDA 2009)*, pages 65–72, Odense, Denmark. Northern European Association for Language Technology (NEALT).

David G. Hays. 1964. Dependency theory: A formalism and some observations. *Language*, 40(4):511.

Johannes Heinecke and Francis M. Tyers. 2019. Development of a Universal Dependencies treebank for Welsh. In *Proceedings of the Celtic Language Technology Workshop*, pages 21–31, Dublin, Ireland. European Association for Machine Translation.

Benjamin Heinzerling and Michael Strube. 2018. BPEmb: Tokenization-free pre-trained subword embeddings in 275 languages. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*.

One-Soon Her and Chen-Tien Hsieh. 2010. On the semantic distinction between classifiers and measure words in chinese. *Language and linguistics*, 11(3):527–551.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Liang Huang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1077–1086, Uppsala, Sweden. Association for Computational Linguistics.

Richard Hudson. 2017. Cross-language diversity, head-direction and grammars. comment on "Dependency distance: A new perspective on syntactic patterns in natural languages" by Haitao Liu et al. *Physics of life reviews*, 21:204–206.

Rebecca Hwa, Philip Resnik, Amy Weinberg, Clara Cabezas, and Okan Kolak. 2005. Bootstrapping parsers via syntactic projection across parallel texts. *Natural Language Engineering*, 11(3):311–325.

Timo Järvinen and Pasi Tapanainen. 1998. Towards an implementable dependency grammar. In *Processing of Dependency-Based Grammars*.

Tao Ji, Yuanbin Wu, and Man Lan. 2019. Graph-based dependency parsing with graph neural networks. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2475–2485, Florence, Italy. Association for Computational Linguistics.

Jingyang Jiang and Haitao Liu. 2015. The effects of sentence length on dependency distance, dependency direction and the implications–based on a parallel english–chinese dependency treebank. *Language Sciences*, 50:93–104.

Anders Johannsen, Željko Agić, and Anders Søgaard. 2016. Joint part-of-speech and dependency projection from multiple sources. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 561–566.

Jaap Jumelet and Dieuwke Hupkes. 2018. Do language models understand anything? On the ability of LSTMs to understand negative polarity items. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 222–231.

Marcin Junczys-Dowmunt, Roman Grundkiewicz, Tomasz Dwojak, Hieu Hoang, Kenneth Heafield, Tom Neckermann, Frank Seide, Ulrich Germann, Alham Fikri Aji, Nikolay Bogoychev, et al. 2018. Marian: Fast neural machine translation in C++. In *Proceedings of ACL 2018, System Demonstrations*, pages 116–121.

Jenna Kanerva, Filip Ginter, Niko Miekka, Akseli Leino, and Tapio Salakoski. 2018. Turku neural parser pipeline: An end-to-end system for the CoNLL 2018 shared task. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 133–142, Brussels, Belgium. Association for Computational Linguistics.

Katharina Kann, Ophélie Lacroix, and Anders Søgaard. 2020. Weakly supervised POS taggers perform poorly on truly low-resource languages. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(5):8066–8073.

Yova Kementchedjhieva, Mark Anderson, and Anders Søgaard. 2021. John praised Mary because *he*? Implicit causality bias and its interaction with explicit cues in LMs. In *Findings of the Association for Computational Linguistics: ACL 2021*, Online. Association for Computational Linguistics.

Yoon Kim and Alexander M Rush. 2016. Sequence-level knowledge distillation. In *Proceedings of EMNLP*, pages 1317–1327.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional LSTM feature representations. *Transactions of the Association for Computational Linguistics*, 4:313–327.

HE Klockmann. 2012. Polish numerals and quantifiers: A syntactic analysis of subject-verb agreement mismatches. Master's thesis, Utrecht University.

Dan Kondratyuk and Milan Straka. 2019. 75 languages, 1 model: Parsing Universal Dependencies universally. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2779–2795, Hong Kong, China. Association for Computational Linguistics.

Sandra Kübler, Ryan McDonald, and Joakim Nivre. 2009. Dependency parsing. In Graeme Hirst, editor, *Synthesis Lectures on Human Language Technologies*, volume 2. Morgan & Claypool.

Sandra Kübler, Ines Rehbein, and Josef van Genabith. 2008. A testsuite for testing parser performance on complex german grammatical constructions. *LOT Occasional Series*, 12:15–28.

Taku Kudo and Yuji Matsumoto. 2002. Japanese dependency analysis using cascaded chunking. In *COLING-02 proceedings of the 6th conference on Natural language learning - Volume 20*, pages 1–7.

Marco Kuhlmann, Carlos Gómez-Rodríguez, and Giorgio Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 673–682, Portland, Oregon, USA. Association for Computational Linguistics.

Artur Kulmizev, Miryam de Lhoneux, Johannes Gontrum, Elena Fano, and Joakim Nivre. 2019. Deep contextualized word embeddings in transition-based and graph-based dependency parsing - a tale of two parsers revisited. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2755–2768, Hong Kong, China. Association for Computational Linguistics.

Jonathan K. Kummerfeld, David Hall, James R. Curran, and Dan Klein. 2012. Parser showdown at the Wall Street corral: An empirical investigation of error types in parser output. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1048–1059, Jeju Island, Korea. Association for Computational Linguistics.

Adhiguna Kuncoro, Miguel Ballesteros, Lingpeng Kong, Chris Dyer, and Noah A. Smith. 2016. Distilling an ensemble of greedy dependency parsers into one MST parser. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1744–1753, Austin, Texas. Association for Computational Linguistics.

Mucahit Kutlu and Ilyas Cicekli. 2016. Noun phrase chunking for turkish using a dependency parser. In *Information Sciences and Systems 2015*, pages 381–391. Springer.

Ophélie Lacroix. 2018. Investigating NP-Chunking with Universal Dependencies for English. In *Proceedings of the Second Workshop on Universal Dependencies (UDW 2018)*, pages 85–90.

Yair Lakretz, German Kruszewski, Theo Desbordes, Dieuwke Hupkes, Stanislas Dehaene, and Marco Baroni. 2019. The emergence of number and syntax units in LSTM language models. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 11–20.

Alberto Lavelli. 2012. An ensemble model for the evalita 2011 dependency parsing task. In *International Workshop on Evaluation of Natural Language and Speech Tool for Italian*, pages 30–36. Springer.

Yann LeCun, John S Denker, and Sara A Solla. 1990. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605.

Iksop Lee and Robert Ramsey. 2000. *The Korean Language*. State University of New York Press, New York.

Miryam de Lhoneux, Yan Shao, Ali Basirat, Eliyahu Kiperwasser, Sara Stymne, Yoav Goldberg, and Joakim Nivre. 2017a. From raw text to universal dependencies - Look, no tags! In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 207–217.

Miryam de Lhoneux, Sara Stymne, and Joakim Nivre. 2017b. Arc-hybrid non-projective dependency parsing with a static-dynamic oracle. In *Proceedings of the 15th International Conference on Parsing Technologies*, pages 99–104, Pisa, Italy. Association for Computational Linguistics.

Miryam de Lhoneux, Sara Stymne, and Joakim Nivre. 2017c. Old school vs. new school: Comparing transition-based parsers with and without neural network enhancement. In *TLT*, pages 99–110.

Charles N. Li and Sandra A. Thompson. 1981. *Mandarin Chinese: a Functional Reference Grammar*. University of California Press, Berkeley.

Shen Li, Zhe Zhao, Renfen Hu, Wensi Li, Tao Liu, and Xiaoyong Du. 2018a. Analogical reasoning on chinese morphological and semantic relations. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 138–143. Association for Computational Linguistics.

Zhongli Li, Qingyu Zhou, Chao Li, Ke Xu, and Yunbo Cao. 2020. Improving bert with syntax-aware local attention. *arXiv preprint arXiv:2012.15150*.

Zuchao Li, Jiaxun Cai, Shexia He, and Hai Zhao. 2018b. Seq2seq dependency parsing. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 3203–3214, Santa Fe, New Mexico, USA. Association for Computational Linguistics.

Zuchao Li, Shexia He, Zhuosheng Zhang, and Hai Zhao. 2018c. Joint learning of POS and dependencies for multilingual Universal Dependency parsing. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 65–73, Brussels, Belgium. Association for Computational Linguistics.

Wang Ling, Chris Dyer, Alan W Black, and Isabel Trancoso. 2015. Two/too simple adaptations of word2vec for syntax problems. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1299–1304.

Tal Linzen, Emmanuel Dupoux, and Yoav Goldberg. 2016. Assessing the ability of LSTMs to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics*, 4(1):521–535.

Haitao Liu. 2007. Probability distribution of dependency distance. *Glottometrics*, 15:1–12.

Haitao Liu. 2008. Dependency distance as a metric of language comprehension difficulty. *Journal of Cognitive Science*, 9(2):159–191.

Haitao Liu. 2010. Dependency direction as a means of word-order typology: A method based on dependency treebanks. *Lingua*, 120(6):1567–1578.

Haitao Liu, Chunshan Xu, and Junying Liang. 2017. Dependency distance: a new perspective on syntactic patterns in natural languages. *Physics of life reviews*, 21:171–193.

Meichun Liu. 2015. Tense and aspect in mandarin chinese. *The Oxford Handbook of Chinese Linguistics*, pages 274–289.

Yijia Liu, Wanxiang Che, Huaipeng Zhao, Bing Qin, and Ting Liu. 2018. Distilling knowledge for search-based structured prediction. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1393–1402, Melbourne, Australia. Association for Computational Linguistics.

Vincenzo Lombardo and Leonardo Lesmo. 1996. An earley-type recognizer for dependency grammar. In *Proceedings of the 16th Conference on Computational Linguistics - Volume 2*, COLING '96, page 723–728, USA. Association for Computational Linguistics.

Liang Lu, Michelle Guo, and Steve Renals. 2017. Knowledge distillation for small-footprint highway networks. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4820–4824. IEEE.

Xuezhe Ma, Zecong Hu, Jingzhou Liu, Nanyun Peng, Graham Neubig, and Eduard Hovy. 2018. Stack-pointer networks for dependency parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1403–1414.

Mitchell P Marcus and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2).

Marie-Catherine de Marneffe, Timothy Dozat, Natalia Silveira, Katri Haverinen, Filip Ginter, Joakim Nivre, and Christopher D. Manning. 2014. Universal stanford dependencies: A cross-linguistic typology. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, pages 4585–4592.

Rebecca Marvin and Tal Linzen. 2018. Targeted syntactic evaluation of language models. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1192–1202.

Takuya Matsuzaki and Jun'ichi Tsujii. 2008. Comparative parser performance analysis across grammar frameworks through automatic tree conversion using synchronous grammars. In *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*, pages 545–552.

Arya D. McCarthy, Ekaterina Vylomova, Shijie Wu, Chaitanya Malaviya, Lawrence Wolf-Sonkin, Garrett Nicolai, Christo Kirov, Miikka Silfverberg, Sabrina J. Mielke, Jeffrey Heinz, Ryan Cotterell, and Mans Hulden. 2019. The SIGMORPHON 2019 shared task: Morphological analysis in context and cross-lingual transfer for inflection. In *Proceedings of the 16th Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 229–244, Florence, Italy. Association for Computational Linguistics.

R. Thomas McCoy, Ellie Pavlick, and Tal Linzen. 2019. Right for the wrong reasons: Diagnosing syntactic heuristics in natural language inference. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3428–3448.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 91–98, Ann Arbor, Michigan. Association for Computational Linguistics.

Ryan McDonald and Joakim Nivre. 2011. Analyzing and integrating dependency parsers. *Computational Linguistics*, 37(1):197–230.

Ryan McDonald, Joakim Nivre, Yvonne Quirmbach-Brundage, Yoav Goldberg, Dipanjan Das, Kuzman Ganchev, Keith Hall, Slav Petrov, Hao Zhang, Oscar Täckström, Claudia Bedini, Núria Bertomeu Castelló, and Jungmee Lee. 2013. Universal Dependency annotation for multilingual parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 92–97, Sofia, Bulgaria. Association for Computational Linguistics.

Ryan McDonald and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *11th Conference of the European Chapter of the Association for Computational Linguistics*, Trento, Italy. Association for Computational Linguistics.

Ryan McDonald, Slav Petrov, and Keith Hall. 2011. Multi-source transfer of delexicalized dependency parsers. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 62–72.

Ryan McDonald and Giorgio Satta. 2007. On the complexity of non-projective data-driven dependency parsing. In *Proceedings of the Tenth International Conference on Parsing Technologies*, pages 121–132, Prague, Czech Republic. Association for Computational Linguistics.

R. S. McGregor. 1977. *Outline of Hindi Grammar*. Oxford University Press, Delhi. 2nd edition.

Sebastian J. Mielke, Ryan Cotterell, Kyle Gorman, Brian Roark, and Jason Eisner. 2019. What kind of language is hard to language-model? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4975–4989, Florence, Italy. Association for Computational Linguistics.

Simon Mille, Alicia Burga, Gabriela Ferraro, and Leo Wanner. 2012. How does the granularity of an annotation scheme influence dependency parsing performance. In *Proceedings of COLING 2012: Posters*, pages 839–852.

George A Miller. 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, 63(2):81.

Yugo Murawaki. 2019. On the definition of Japanese word. *arXiv preprint arXiv:1906.09719*.

Dat Quoc Nguyen and Karin Verspoor. 2018. An improved neural network model for joint POS tagging and dependency parsing. *CoNLL 2018*, page 81.

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57, Barcelona, Spain. Association for Computational Linguistics.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553.

Joakim Nivre. 2009. Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 351–359.

Joakim Nivre, Mitchell Abrams, Željko Agić, Lars Ahrenberg, et al. 2018. Universal Dependencies 2.3. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

Joakim Nivre, Mitchell Abrams, Željko Agić, et al. 2019. Universal Dependencies 2.4. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

Joakim Nivre, Željko Agić, Lars Ahrenberg, et al. 2017. Universal Dependencies 2.1. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2004. Memory-based dependency parsing. In *Proceedings of the Eighth Conference on Computational Natural Language Learning (CoNLL-2004) at HLT-NAACL 2004*, pages 49–56, Boston, Massachusetts, USA. Association for Computational Linguistics.

Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülşen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. 2007. MaltParser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135.

George Orwell. 2002. Politics and the English language. In Peter Davison, editor, *Essays*, pages 954–970. Everyman's Library, New York, London, and Toronto.

Timothy Osborne and Kim Gerdes. 2019. The status of function words in dependency grammar: A critique of universal dependencies (ud). *Glossa*, 4(1):17.

Martha Palmer, Rajesh Bhatt, Bhuvana Narasimhan, Owen Rambow, Dipti Misra Sharma, and Fei Xia. 2009. Hindi syntax: Annotating dependency, lexical predicate-argument structure, and phrase structure. In *The 7th International Conference on Natural Language Processing*, pages 14–17.

Waltraud Paul. 2008. The serial verb construction in chinese: A tenacious myth and a gordian knot. *Linguistic Review - LINGUIST REV*, 25:367–411.

Matthew E. Peters, Mark Neumann, Luke Zettlemoyer, and Wen tau Yih. 2018. Dissecting contextual word embeddings: Architecture and representation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1499–1509.

Slav Petrov, Dipanjan Das, and Ryan McDonald. 2012. A universal part-of-speech tagset. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC-2012)*, pages 2089–2096.

B. Plank and G.J.M. van Noord. 2010. Dutch dependency parser performance across domains. *LOT Occasional Series*, 16:123–138.

Lauma Pretkalniņa and Laura Rituma. 2014. Constructions in latvian treebank: the impact of annotation decisions on the dependency parsing performance. *Baltic HLT*, pages 219–226.

Peng Qi, Timothy Dozat, Yuhao Zhang, and Christopher D. Manning. 2018. Universal dependency parsing from scratch. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 160–170, Brussels, Belgium. Association for Computational Linguistics.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8).

Lance A Ramshaw and Mitchell P Marcus. 1999. Text chunking using transformation-based learning. In *Natural language processing using very large corpora*, pages 157–176. Springer.

G. J. Ramstedt. 1968. *A Korean Grammar*, volume 82 of *Memoirs of the Finno-Ugric Society*. Anthropological Publications, Oosterhout, The Netherlands.

Vikas Raunak. 2017. Simple and effective dimensionality reduction for word embeddings. *Proceedings of NIPS LLD Workshop*.

Ines Rehbein, Julius Steen, Bich-Ngoc Do, and Anette Frank. 2017. Universal Dependencies are hard to parse–or are they? In *Proceedings of the Fourth International Conference on Dependency Linguistics (Depling 2017)*, pages 218–228.

Nils Reimers and Iryna Gurevych. 2017. Reporting score distributions makes a difference: Performance study of LSTM-networks for sequence tagging. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 338–348.

Frank Rosenblatt. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. 1998. A metric for distributions with applications to image databases. In *Proceedings of the Sixth International Conference on Computer Vision*, ICCV '98, pages 59–, Washington, DC, USA. IEEE Computer Society.

Devendra Sachan, Yuhao Zhang, Peng Qi, and William L. Hamilton. 2021. Do syntax trees help pre-trained transformers extract information? In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 2647–2661, Online. Association for Computational Linguistics.

Kenji Sagae, Yusuke Miyao, Rune Saetre, and Jun'ichi Tsujii. 2008. Evaluating the effects of treebank size in a practical application for parsing. In *Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, pages 14–20, Columbus, Ohio. Association for Computational Linguistics.

Kenji Sagae and Jun-ichi Tsujii. 2010. Dependency parsing and domain adaptation with data-driven lr models and parser ensembles. *Trends in Parsing Technology*, pages 57–68.

Erik F. Tjong Kim Sang. 2000. Transforming a chunker to a parser. In *CLIN*, pages 177–188.

Bharat Bhusan Sau and Vineeth N Balasubramanian. 2016. Deep model compression: Distilling knowledge from noisy teachers. *arXiv preprint arXiv:1610.09650*.

Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681.

Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. 2019. Green AI. *arXiv preprint arXiv:1907.10597*.

Abigail See, Minh-Thang Luong, and Christopher D Manning. 2016. Compression of neural machine translation models via pruning. In *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*, pages 291–301.

P. K. Sen, Richard H. Lindeman, Peter F. Merenda, and Ruth Z. Gold. 1981. Introduction to bivariate and multivariate analysis. *Journal of the American Statistical Association*, 76(375):752.

S. S. Shapiro and M. B. Wilk. 1965. An analysis of variance test for normality (complete samples). *Biometrika*, 52:591–611.

Tianze Shi, Felix G. Wu, Xilun Chen, and Yao Cheng. 2017. Combining global models for parsing universal dependencies. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 31–39, Vancouver, Canada. Association for Computational Linguistics.

Anna Siewierska. 1993. Syntactic weight vs. information structure and word order variation in polish. *Journal of Linguistics*, 29:233–265.

Koustuv Sinha, Robin Jia, Dieuwke Hupkes, Joelle Pineau, Adina Williams, and Douwe Kiela. 2021. Masked language modeling and the distributional hypothesis: Order word matters pre-training for little.

Aaron Smith, Bernd Bohnet, Miryam de Lhoneux, Joakim Nivre, Yan Shao, and Sara Stymne. 2018. 82 treebanks, 34 models: Universal dependency parsing with multi-treebank models. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 113–123.

Aaron Smith, Miryam de Lhoneux, Sara Stymne, and Joakim Nivre. 2018. An investigation of the interactions between pre-trained word embeddings, character models and POS tags in dependency parsing. In *EMNLP 2018: 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2711–2720.

R. Snell and S. Weightman. 1989. *Teach yourself Hindi*. Hodder and Stoughton, London.

Richard Socher, Christopher D Manning, and Andrew Y Ng. 2010. Learning continuous phrase representations and syntactic parsing with recursive neural networks. In *Proceedings of the NIPS-2010 deep learning and unsupervised feature learning workshop*, volume 2010, pages 1–9.

Anders Søgaard. 2020. Some languages seem easier to parse because their treebanks leak. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2765–2770, Online. Association for Computational Linguistics.

Anders Søgaard, Jasmijn Bastings, Katja Filippova, and Sebastian Ebert. 2021. We need to talk about random splits. In *Proceedings of the 16th conference of the European Chapter of the Association for Computational Linguistics (to appear)*, Online.

Anders Søgaard and Yoav Goldberg. 2016. Deep multi-task learning with low level tasks supervised at lower layers. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 231–235, Berlin, Germany. Association for Computational Linguistics.

Ho-Min Sohn. 1999. *The Korean Language*. Cambridge University Press, Cambridge.

Seok Choong Song. 1988. *Explorations in Korean Syntax and Semantics*. Institute of East Asian Studies, University of California Berkeley, Berkeley.

Ionut-Teodor Sorodoc, Kristina Gulordava, and Gemma Boleda. 2020. Probing for referential information in language models. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4177–4189.

Drahomíra Johanka Spoustová and Miroslav Spousta. 2010. Dependency parsing as a sequence labeling task. *The Prague Bulletin of Mathematical Linguistics*, 94(2010):7–14.

Rachele Sprugnoli, Marco Passarotti, Flavio Massimiliano Cecchini, and Matteo Pellegrini. 2020. Overview of the EvaLatin 2020 evaluation campaign. In *Proceedings of LT4HALA 2020 - 1st Workshop on Language Technologies for Historical and Ancient Languages*, pages 105–110, Marseille, France. European Language Resources Association (ELRA).

Pontus Stenetorp. 2013. Transition-based dependency parsing using recursive neural networks.

Milan Straka. 2018. Udpipe 2.0 prototype at conll 2018 ud shared task. *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 197–207.

Milan Straka, Jan Hajic, and Jana Straková. 2016. Udpipe: Trainable pipeline for processing conll-u files performing tokenization, morphological analysis, pos tagging and parsing. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 4290–4297.

Milan Straka, Jan Hajič, Jana Straková, and Jan Hajič jr. 2015. Parsing universal dependency treebanks using neural networks and search-based oracle. In *Proceedings of Fourteenth International Workshop on Treebanks and Linguistic Theories (TLT 14)*.

Milan Straka and Jana Straková. 2017. Tokenizing, POS tagging, Lemmatizing and Parsing UD 2.0 with UDPipe. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 88–99.

Milan Straka and Jana Straková. 2019a. Universal Dependencies 2.4 models for UDPipe. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

Milan Straka and Jana Straková. 2019b. Universal dependencies 2.5 models for UDPipe (2019-12-06). LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*.

Michalina Strzyz, David Vilares, and Carlos Gómez-Rodríguez. 2019a. Sequence labeling parsing by learning across representations. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 5350–5357. Association for Computational Linguistics.

Michalina Strzyz, David Vilares, and Carlos Gómez-Rodríguez. 2019b. Viable dependency parsing as sequence labeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 717–723, Minneapolis, Minnesota. Association for Computational Linguistics.

Michalina Strzyz, David Vilares, and Carlos Gómez-Rodríguez. 2020. Bracketing encodings for 2-planar dependency parsing. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 2472–2484, Barcelona, Spain (Online). International Committee on Computational Linguistics.

Sara Stymne, Miryam de Lhoneux, Aaron Smith, and Joakim Nivre. 2018. Parser training with heterogeneous treebanks. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 619–625, Melbourne, Australia. Association for Computational Linguistics.

Mihai Surdeanu and Christopher D. Manning. 2010. Ensemble models for dependency parsing: Cheap and good? In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 649–652, Los Angeles, California. Association for Computational Linguistics.

Oscar Täckström, Dipanjan Das, Slav Petrov, Ryan McDonald, and Joakim Nivre. 2013. Token and type constraints for cross-lingual part-of-speech tagging. *Transactions of the Association for Computational Linguistics*, 1:1–12.

Aniruddha Tammewar, Karan Singla, Bhasha Agrawal, Riyaz Bhat, and Dipti Misra Sharma. 2015. Can distributed word embeddings be an alternative to costly linguistic features: A study on parsing hindi. In *Proceedings of the 6th Workshop on Statistical Parsing of Morphologically Rich Languages (SPMRL 2015)*, pages 21–30.

Takaaki Tanaka, Katsuhiko Hayashi, and Masaaki Nagata. 2017. Hierarchical word structure-based parsing: A feasibility study on UD-style dependency parsing in Japanese. In *Proceedings of the 15th International Conference on Parsing Technologies*, pages 56–60, Pisa, Italy. Association for Computational Linguistics.

Raphael Tang, Yao Lu, Linqing Liu, Lili Mou, Olga Vechtomova, and Jimmy Lin. 2019. Distilling task-specific knowledge from BERT into simple neural networks. *arXiv preprint arXiv:1903.12136*.

Pasi Tapanainen and Timo Jarvinen. 1997. A non-projective dependency parser. In *Fifth Conference on Applied Natural Language Processing*, pages 64–71, Washington, DC, USA. Association for Computational Linguistics.

David Temperley and Daniel Gildea. 2018. Minimizing syntactic dependency lengths: typological/cognitive universal? *Annual Review of Linguistics*, 4:67–80.

Ian Tenney, Dipanjan Das, and Ellie Pavlick. 2019. BERT rediscovers the classical NLP pipeline. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4593–4601.

Lucien Tesnière. 1959. Éléments de syntaxe structurale.

Jörg Tiedemann. 2015. Cross-lingual dependency parsing with universal dependencies and predicted pos labels. *Proceedings of the Third International Conference on Dependency Linguistics (Depling 2015)*, pages 340–349.

U. Timm. 1969. Coherent bremsstrahlung of electrons in crystals. *Protein Science*, 17(12):765–808.

Ivan Titov and James Henderson. 2007. A latent variable model for generative dependency parsing. In *Proceedings of the Tenth International Conference on Parsing Technologies*, pages 144–155, Prague, Czech Republic. Association for Computational Linguistics.

Shisanu Tongchim, Virach Sornlertlamvanich, and Hitoshi Isahara. 2008. Experiments in base-NP chunking and its role in dependency parsing for Thai. In *Coling 2008: Companion volume: Posters*, pages 123–126, Manchester, UK. Coling 2008 Organizing Committee.

Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 173–180. Association for computational Linguistics.

Yoshimasa Tsuruoka and Jun'ichi Tsujii. 2005. Chunk parsing revisited. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 133–140, Vancouver, British Columbia. Association for Computational Linguistics.

Ahmet Üstün, Arianna Bisazza, Gosse Bouma, and Gertjan van Noord. 2020. UDapter: Language adaptation for truly Universal Dependency parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2302–2315, Online. Association for Computational Linguistics.

Raphael Vallat. 2018. Pingouin: statistics in python. *Journal of Open Source Software*, 3(31):1026.

Clara Vania, Yova Kementchedjhieva, Anders Søgaard, and Adam Lopez. 2019. A systematic comparison of methods for low-resource dependency parsing on genuinely low-resource languages. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1105–1116.

Leonid Nisonovich Vaserstein. 1969. Markov processes over denumerable products of spaces, describing large systems of automata. *Problemy Peredachi Informatsii*, 5(3):64–72.

David Vilares, Mostafa Abdou, and Anders Søgaard. 2019. Better, faster, stronger sequence tagging constituent parsers. *arXiv preprint arXiv:1902.10985*.

David Vilares, Michalina Strzyz, Anders Søgaard, and Carlos Gómez-Rodríguez. 2020. Parsing as pretraining. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9114–9121.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *NIPS'15 Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, volume 28, pages 2692–2700.

Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. 2019. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5797–5808.

Atro Voutilainen. 1998. Does tagging help parsing? A case study on finite state parsing. In *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing*, pages 25–36. Association for Computational Linguistics.

Weishui Wan, Shingo Mabu, Kaoru Shimada, Kotaro Hirasawa, and Jinglu Hu. 2009. Enhancing the generalization ability of neural networks through controlling the hidden layers. *Applied Soft Computing*, 9(1):404–414.

Xinyu Wang, Jingxian Huang, and Kewei Tu. 2019. Second-order semantic dependency parsing with end-to-end neural networks. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4609–4618, Florence, Italy. Association for Computational Linguistics.

Xinyu Wang, Yong Jiang, and Kewei Tu. 2020. Enhanced Universal Dependency parsing with second-order inference and mixture of training data. In *Proceedings of the 16th International Conference on Parsing Technologies and the IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies*, pages 215–220, Online. Association for Computational Linguistics.

Alex Warstadt, Amanpreet Singh, and Samuel R. Bowman. 2019. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, 7:625–641.

David Weiss, Chris Alberti, Michael Collins, and Slav Petrov. 2015. Structured training for neural network transition-based parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 323–333, Beijing, China. Association for Computational Linguistics.

Bernd Wiese. 2011. Optimal specifications: On case marking in polish. *Syntax and morphology multidimensional*, 24:101.

Peter Wills and François G. Meyer. 2020. Metrics for graph comparison: A practitioner's guide. *PLOS ONE*, 15(2):1–54.

Alina Wróblewska. 2018. Extended and enhanced polish dependency bank in universal dependencies format. In *Proceedings of the Second Workshop on Universal Dependencies (UDW 2018)*, pages 173–182.

Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2016. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828.

Zenan Xu, Daya Guo, Duyu Tang, Qinliang Su, Linjun Shou, Ming Gong, Wanjun Zhong, Xiaojun Quan, Nan Duan, and Daxin Jiang. 2020. Syntax-enhanced pre-trained model. *arXiv preprint arXiv:2012.14116*.

Naiwen Xue, Fei Xia, Fu-Dong Chiou, and Marta Palmer. 2005. The penn chinese treebank: Phrase structure annotation of a large corpus. *Natural language engineering*, 11(2):207.

Nianwen Xue, Fu-Dong Chiou, and Martha Palmer. 2002. Building a large-scale annotated chinese corpus. In *COLING 2002: The 19th International Conference on Computational Linguistics*.

Jie Yang and Yue Zhang. 2018. NCRF++: An open-source neural sequence labeling toolkit. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*.

Liner Yang, Meishan Zhang, Yang Liu, Maosong Sun, Nan Yu, and Guohong Fu. 2017. Joint POS tagging and dependence parsing with transition-based neural networks. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 26(8):1352–1358.

Bee Wah Yap and C. H. Sim. 2011. Comparisons of various types of normality tests. *Journal of Statistical Computation and Simulation*, 81(12):2141–2155.

Anssi Yli-Jyrä and Carlos Gómez-Rodríguez. 2017. Generic axiomatization of families of noncrossing graphs in dependency parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1745–1755, Vancouver, Canada. Association for Computational Linguistics.

Seunghak Yu, Nilesh Kulkarni, Haejun Lee, and Jihie Kim. 2018. On-device neural language model based word prediction. In *Proceedings of the 27th International Conference on Computational Linguistics: System Demonstrations*, pages 128–131.

Xiyu Yu, Tongliang Liu, Xinchao Wang, and Dacheng Tao. 2017. On compressing deep models by low rank and sparse decomposition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7370–7379.

Daniel Zeman. 2008. Reusable tagset conversion using tagset drivers. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*.

Daniel Zeman, Jan Hajič, Martin Popel, Martin Potthast, Milan Straka, Filip Ginter, Joakim Nivre, and Slav Petrov. 2018. CoNLL 2018 shared task: Multilingual parsing from raw text to universal dependencies. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 1–21, Brussels, Belgium. Association for Computational Linguistics.

Daniel Zeman, Joakim Nivre, et al. 2020. Universal Dependencies 2.6. LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

Daniel Zeman, Martin Popel, Milan Straka, Jan Hajic, et al. 2017. Conll 2017 shared task: Multilingual parsing from raw text to universal dependencies. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 1–19, Vancouver, Canada. Association for Computational Linguistics.

Daniel Zeman and Philip Resnik. 2008. Cross-language parser adaptation between related languages. In *Proceedings of the IJCNLP-08 Workshop on NLP for Less Privileged Languages*, pages 35–42.

Meishan Zhang, Yue Zhang, Wanxiang Che, and Ting Liu. 2014. Character-level Chinese dependency parsing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1326–1336, Baltimore, Maryland. Association for Computational Linguistics.

Xiang Zhang and Yann LeCun. 2015. Text understanding from scratch. *arXiv preprint arXiv:1502.01710*.

Xingxing Zhang, Jianpeng Cheng, and Mirella Lapata. 2017. Dependency parsing as head selection. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 665–676, Valencia, Spain. Association for Computational Linguistics.

Yi Zhang and Rui Wang. 2009. Correlating natural language parser performance with statistical measures of the text. In *KI'09 Proceedings of the 32nd annual German conference on Advances in artificial intelligence*, pages 217–224.

Yu Zhang, Zhenghua Li, and Min Zhang. 2020a. Efficient second-order TreeCRF for neural dependency parsing. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 3295–3305, Online. Association for Computational Linguistics.

Yu Zhang, Zhenghua Li, Houquan Zhou, and Min Zhang. 2020b. Is POS tagging necessary or even helpful for neural dependency parsing? *arXiv preprint arXiv:2003.03204*.

Yuan Zhang and David Weiss. 2016. Stack-propagation: Improved representation learning for syntax. *arXiv preprint arXiv:1603.06598*.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193, Portland, Oregon, USA. Association for Computational Linguistics.

Zhuosheng Zhang, Yuwei Wu, Junru Zhou, Sufeng Duan, Hai Zhao, and Rui Wang. 2020c. Sg-net: Syntax guided transformer for language representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Junru Zhou and Hai Zhao. 2019. Head-driven phrase structure grammar parsing on Penn Treebank. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2396–2408.

Resumen prolongado en castellano

Introducción La temática de esta tesis se centra en el análisis sintáctico de dependencias. El análisis sintáctico de dependencias es el acto de convertir datos lingüísticos (oraciones en lenguaje humano) en una estructura sintáctica formal, concretamente, un árbol de dependencias definido por una gramática de dependencias.

El trabajo presentado aquí se puede dividir de forma natural en dos partes. La primera cubre los intentos de desarrollar nuevos sistemas para el análisis de dependencias. Este trabajo se realizó en el marco del equipo de investigación del proyecto FastParse. Era un nombre adecuado, porque tratamos de desarrollar técnicas y sistemas que fueran eficientes. En este contexto, entendemos por eficiencia que el sistema sea rápido (en el momento de la inferencia) manteniendo al mismo tiempo una precisión razonable. La primera técnica que probamos fue influenciada por la psicolingüística. Se basa en las restricciones de la memoria de trabajo del cerebro humano y la necesidad hipotética de que los humanos creen representaciones jerárquicas abstractas de la entrada lingüística. Esto se hace con el fin de no perder información, y a este modelo de la comprensión humana del lenguaje se le llama procesamiento Chunk-and-Pass. La técnica en sí fue un fracaso abyecto, pero parte del trabajo auxiliar que se llevó a cabo para implementarla fue útil e interesante. El segundo método que usamos para desarrollar analizadores eficientes fue más exitoso. Básicamente, utilizamos modelos más grandes y con más parámetros para ayudar a guiar modelos con menos parámetros (y por lo tanto, más rápidos) utilizando una técnica de destilación de modelos. Esto dio como resultado el sistema de análisis sintáctico moderno más rápido, que además es también más preciso que el siguiente sistema de análisis sintáctico más rápido.

La segunda parte de la tesis se centra en la evaluación de analizadores sintácticos. El primer capítulo de esta parte describe un trabajo centrado en el desplazamiento de dependencia entre *tokens* de una oración. Dicho desplazamiento se define como la distancia dirigida (con signo) entre una palabra y su padre (la palabra de la que depende en el árbol de dependencias). Usamos esta medida como base de dos análisis similares. El primero compara la distribución inherente de los desplazamientos de dependencia hacia la que están sesgados ciertos algoritmos basados en transiciones con las que se encuentran en los corpus. Observamos una relación entre la similitud entre estas distribuciones y el rendimiento de los algoritmos en cada corpus dado. De manera similar, comparamos las distribuciones de los datos de entrenamiento y prueba y encontramos una fuerte correlación entre su similitud y el rendimiento del análisis sintáctico, incluso cuando se tienen en cuenta las covariantes y dos paradigmas diferentes de sistemas de análisis. A continuación, evaluamos las etiquetas morfosintácticas (part-of-speech tags) y cómo la precisión de los etiquetadores afecta a la utilidad de estas etiquetas para los sistemas de análisis. Aquí ofrecemos una evaluación exhaustiva tanto para un analizador basado en grafos como para un analizador basado en transiciones, encontrando que utilizar etiquetas predichas por un etiquetador automático generalmente perjudica el rendimiento en comparación con no usar ninguna etiqueta en absoluto. Extendemos este análisis para evaluar si los analizadores sintácticos aprenden algo intrínsecamente sobre los tipos de palabras, haciendo que la información de estas etiquetas sea redundante o potencialmente conflictiva.

Obtenemos resultados que sugieren que los analizadores aprenden algo en esta dirección, y que lo que no aprenden coincide en gran medida con lo que los etiquetadores no logran capturar. También ampliamos este análisis para analizar específicamente la utilidad de las etiquetas morfosintácticas en contextos de bajos recursos, con base en los hallazgos del análisis original. Aquí observamos que los corpus más pequeños pueden aprovechar más fácilmente, al menos en cierto grado, la información contenida en las etiquetas predichas, incluso cuando la precisión de los etiquetadores no es particularmente alta.

La introducción se podría resumir como una formulación de la tesis en términos de un par de preguntas:

- P1. ¿Podemos mejorar la eficiencia de los analizadores sintácticos modernos con respecto a la precisión y la velocidad del análisis?
- P2. ¿Podemos explicar al menos parcialmente la variación en el rendimiento del análisis observada en ciertos contextos?

Parte I: Análisis sintáctico de dependencias En esta parte, se introduce el análisis sintáctico de dependencias. Esto incluye una descripción de la gramática de dependencias, la tarea del análisis sintáctico de dependencias, los diferentes métodos que se utilizan para el análisis y una discusión sobre el trabajo relacionado. El análisis sintáctico de dependencias es el acto de convertir datos lingüísticos en una estructura sintáctica formal, específicamente un árbol de dependencias definido por una gramática de dependencias.

Capítulo 1: Preliminares Los preliminares dan detalles sobre la gramática de dependencias y el análisis sintáctico de dependencias. Primero se ofrece una breve descripción general de las gramáticas de dependencias, contrastándolas con la alternativa que proporcionan las gramáticas de constituyentes. Después, se proporciona una descripción más detallada del formalismo de gramática de dependencias que se utiliza principalmente a lo largo de esta tesis: las Dependencias Universales (UD), que proporcionan unos criterios de anotación que permiten representar la sintaxis de diferentes idiomas y familias lingüísticas de manera uniforme. A continuación, se presentan las dos familias más destacadas de métodos que se utilizan para el análisis de dependencias: análisis basado en transiciones, donde el analizador es una máquina de estados que construye dependencias a medida que transiciona de unos estados a otros, y análisis basado en grafos, donde a cada posible análisis de la entrada se le asocia una puntuación que se obtiene agregando puntuaciones de subanálisis más pequeños (a menudo, aunque no siempre, dependencias individuales), y el proceso de análisis consiste en la búsqueda del análisis de puntuación máxima. También se presentan los algoritmos concretos de ambos métodos que se utilizan a lo largo de la tesis.

Los detalles presentados en estos capítulos deberían ser información suficiente para comprender cada uno de los capítulos siguientes de forma aislada para la mayoría de los lectores interesados en esta tesis. La descripción de la gramática de dependencias y UD es suficiente para seguir el trabajo, y cuando se requiere más información para una parte específica, la describimos en los capítulos respectivos. Los detalles sobre los diferentes sistemas de análisis sintáctico dan una idea amplia del análisis sintáctico de dependencias en PLN, que se ampliará en el capítulo siguiente, pero con una descripción menos formal a medida que analizamos el trabajo reciente en este campo.

Capítulo 2: Trabajo relacionado En esta sección se discute el trabajo reciente sobre análisis de dependencias. Esto incluye tanto las técnicas y sistemas utilizados antes de que las redes neuronales se convirtieran en el estándar de facto en cuanto a métodos de aprendizaje automático en PLN, como los sistemas basados en redes neuronales y las técnicas actuales. Se hace un recorrido por distintas técnicas que actualmente tienen éxito en analizadores de dependencias. Asimismo, se analiza la situación actual con respecto al rendimiento del análisis sintáctico y también con respecto a la velocidad de los analizadores. Por último, se

discute la relación de los modelos de lenguaje preentrenados con el análisis sintáctico y su influencia en la vigencia de esta tarea.

Parte II: Desarrollo de analizadores sintácticos En esta parte se describe el trabajo sobre el desarrollo de sistemas de análisis sintáctico, que constituye el primer bloque de las contribuciones novedosas que se presentan en esta tesis.

Capítulo 3: Chunk-and-Pass En este capítulo, evaluamos la tarea de identificación de frases cortas o fragmentos de texto (chunking) y su interacción con el análisis sintáctico. Usamos un algoritmo evolutivo para elegir el mejor conjunto de reglas utilizadas para extraer fragmentos a partir de un corpus, y evaluamos su impacto en el etiquetado morfosintáctico (part-of-speech tagging), el etiquetado de características morfológicas (morphological feature tagging) y el análisis sintáctico de dependencias. Los resultados permiten concluir que estos fragmentos extraídos automáticamente son muy útiles para estas tareas, en especial cuando se utilizan en una configuración de aprendizaje multitarea. Luego, presentamos una forma de extraer fragmentos utilizando la teoría de la información. Usamos *normalised point-wise mutual information* (NPMI) para clasificar las reglas potenciales en función de las etiquetas morfosintácticas de cada *token*, así como la etiqueta de su padre y la relación que las conecta.

A continuación, introducimos el análisis sintáctico mediante modelos Chunk-and-Pass, y evaluamos su eficacia. En realidad, los resultados de este experimento solo se pueden describir como un fracaso abyecto con respecto al objetivo subyacente. Sin embargo, el sistema de etiquetado NPMI podría ser potencialmente útil para generar características para otros sistemas y es infinitamente más rápido que usar una técnica evolutiva. Más allá de eso, tuvimos que evaluar tres sistemas de análisis principales de forma exhaustiva y coherente. Esto ha dado como resultado una imagen muy clara de los analizadores sintácticos modernos: los analizadores sintácticos Biaffine deberían ser realmente la opción predeterminada, salvo en situaciones en las que existan límites de tiempo muy restrictivos (y se esté restringido a utilizar CPU), en cuyo caso los sistemas de etiquetado de secuencias son una opción viable.

Capítulo 4: Destilación Hemos demostrado que la destilación puede producir modelos eficientes, que superan en velocidad a los modelos de partida más grandes, y son además más precisos si se comparan con analizadores sintáticos normales de igual tamaño. Esto da como resultado analizadores que son los más rápidos entre los analizadores modernos y más precisos que la siguiente variante más rápida de analizadores existente en la actualidad.

Sin embargo, en un segundo experimento donde intentamos extender este resultado al análisis con Dependencias Universales Mejoradas (EUD), obtuvimos resultados que no corroboran este hallazgo. Los modelos destilados no superaron a sus modelos de referencia de tamaño equivalente. Esto quizás se deba a una serie de diferencias en el contexto experimental, pero es probable que se explique principalmente por no usar etiquetas morfosintácticas como características de entrada, mientras que en el resultado anterior se utilizaron de esta manera etiquetas morfosintácticas anotadas por humanos (gold standard). Esta cuestión requiere un análisis más detallado. Este trabajo también destacó el alto coste de entrenamiento que añade la destilación en comparación con el entrenamiento normal de analizadores sintácticos. Esto significa que el analizador debe usarse en una gran cantidad de datos para que el ahorro de energía a lo largo de su uso en producción llegue a compensar esta diferencia inicial de gasto energético en el entrenamiento, lo cual resalta el problema de considerar cómo se utilizarán los analizadores en producción cuando se analiza su eficiencia.

A pesar de centrarse en la eficiencia, nuestra presentación oficial a la tarea compartida IWPT 2020 obtuvo un *enhanced labelled attachment score* (ELAS) promedio de 74,04, que fue el cuarto mejor sistema de 9 presentaciones completas. Nuestra puntuación mejorada después de entrenar modelos destilados hasta la convergencia (o más cerca de la convergencia) obtuvo una puntuación media de 76,14.

Parte III: Evaluación de analizadores sintácticos En esta parte se describe el trabajo sobre evaluación de analizadores sintácticos de dependencias, analizando factores que condicionan y explican parcialmente el rendimiento de diferentes algoritmos de análisis en distintos corpus.

Capítulo 5: Desplazamiento de dependencia En este capítulo, introducimos el concepto de distribución sobre los desplazamientos de dependencia. Mostramos que cada algoritmo basado en transiciones está intrínsecamente sesgado hacia una determinada distribución de este tipo, según las transiciones disponibles. Luego mostramos que la similitud entre estas distribuciones y las distribuciones encontradas en los datos de prueba de los corpus está relacionada con el rendimiento del análisis. Para hacerlo, tuvimos que realizar un análisis con agrupamiento según la longitud de las oraciones, ya que la correlación no fue observable al considerar la muestra de datos como un todo.

A continuación, tomamos este concepto y lo aplicamos a la distancia entre las distribuciones que se ven en los datos de entrenamiento y prueba, llamándola *edge displacement Vaserstein distance* (EDV). Encontramos una señal bastante fuerte en los datos entre EDV y *labelled attachment score* (LAS), incluso cuando se tienen en cuenta las covariables. También mostramos que la señal es más fuerte cuando se realiza un análisis de agrupamiento de la longitud de las oraciones, pero que la correlación aún es bastante clara tomando la muestra de datos completa. Asimismo, mostramos que podríamos llevar a cabo un procedimiento de muestreo para obtener divisiones adversas y complementarias de los datos, y de este modo derivar límites empíricos del rendimiento de los analizadores sintácticos que puedan proporcionar una medida más robusta de su calidad.

Capítulo 6: Etiquetas morfosintácticas Realizamos una serie de experimentos controlados en los que modificamos la precisión de los etiquetadores morfosintácticos (part-of-speech taggers) que se utilizan para guiar a los analizadores sintácticos. Observamos una tendencia lineal para dos sistemas de análisis moderno de diferentes tipos, y descubrimos que usar etiquetas predichas era casi exclusivamente peor que no usar etiquetas. Además, observamos un aumento no lineal en el rendimiento del análisis cuando se utilizan etiquetas "gold standard" (anotadas por humanos), lo que sugiere algún tipo de excepcionalidad. También obtuvimos resultados provisionales que sugerían que los corpus más pequeños aún pueden beneficiarse del uso de etiquetas predichas provenientes de etiquetadores inexactos.

Investigamos la mencionada excepcionalidad analizando lo que los analizadores sintácticos aprenden inherentemente sobre los tipos de palabras, y cómo esto corresponde a lo que los etiquetadores no son capaces de predecir. Descubrimos que hay un fuerte cruce entre lo que los analizadores no aprenden implícitamente y lo que los etiquetadores no consiguen predecir. También destacamos tipos de errores y contextos consistentes en un subconjunto de árboles de UD. Y obtuvimos resultados que sugieren que un etiquetador más complejo sería beneficioso si pudiera aprender cuándo debe hacer predicciones y cuándo no, para evitar enviar señales erróneas a los analizadores sintácticos. El conjunto de corpus utilizado aquí fue diferente al usado en el estudio anterior, y en este análisis sí observamos un aumento moderado en el rendimiento cuando se utilizan etiquetas predichas en lugar de no utilizarlas en absoluto (0,2 LAS).

Finalmente, profundizamos en la cuestión de si los etiquetadores morfosintácticos de baja precisión siguen siendo útiles en contextos de bajos recursos. Descubrimos que ofrecen un pequeño aumento en el rendimiento del análisis (aproximadamente 1 punto de LAS) y, más específicamente, podemos aumentar el rendimiento del análisis si disponemos de más datos anotados solamente con etiquetas morfosintácticas (aunque no se disponga de árboles de dependencias para estos datos).

Conclusión En primer lugar, los resultados de nuestros esfuerzos por desarrollar analizadores sintácticos rápidos y precisos fueron mixtos. La técnica de Chunk-and-Pass

resultó atroz, lo que en retrospectiva podría no ser tan sorprendente. Quizás con un sistema mucho mejor para modelar la identificación de fragmentos, los resultados serían mejores. Pero implicaría aumentar los costes computacionales y, por lo tanto, ralentizar el sistema. Éste es, de hecho, el problema fundamental de esta técnica: cualquier método introducido para optimizar una de las métricas de interés impacta grave y negativamente en la otra. Pero a pesar de que esta línea de investigación no logró producir un analizador sintáctico rápido y preciso, el trabajo realizado para extraer fragmentos de los corpus tiene el potencial de ser útil. Hemos demostrado que pueden aumentar moderadamente la precisión del etiquetado y el análisis, especialmente en una configuración *multitask learning* (MTL). Así que algo bueno salió de ello.

La técnica de destilación tiene resultados contradictorios, sin embargo, el experimento inicial se llevó a cabo en circunstancias ligeramente diferentes. Parece que las etiquetas morfosintácticas son realmente útiles en este contexto, para ayudar al modelo más grande a destilar su conocimiento a la red más pequeña. Los modelos destilados en el *Penn Treebank* (PTB) usaron etiquetas morfosintácticas predichas estándar como características y claramente empujan el límite del frente de Pareto, especialmente el modelo más pequeño en la CPU.

La tesis ofreció alguna explicación sobre las diferencias observadas en el análisis sintáctico con respecto a diferentes algoritmos, y también con respecto a diferencias en los datos. El trabajo que se centra en algoritmos basados en transiciones arrojó algo de luz sobre un problema abierto desde hace mucho tiempo. Mostró que una gran proporción de la variabilidad observada puede explicarse por lo similar que es la distribución del desplazamiento de dependencia en un corpus a la distribución inherente hacia la que está sesgado un algoritmo. Esto requirió llevar a cabo un análisis con agrupamiento por la longitud de las oraciones, que es quizás una clara conclusión general de esta tesis: que aquellos análisis que se centran en, o utilizan, características lingüísticas que están relacionadas con la longitud de las oraciones, casi siempre se benefician de un procedimiento de agrupamiento según la longitud de cada oración. De lo contrario, es fácil caer en resultados espurios o que fenómenos significativos permanezcan ocultos. La segunda parte de este análisis se centró en la similitud de estas distribuciones entre los datos de entrenamiento y de prueba de los corpus. Observamos una fuerte correlación incluso al controlar las covariantes y discutimos el uso práctico de este hallazgo. Hicimos una demostración práctica de su uso para guiar un procedimiento de muestreo contradictorio (adversarial sampling) con el fin de evaluar más a fondo los analizadores sintácticos.

Por último, investigamos la utilidad de las etiquetas morfosintácticas para los analizadores en varios contextos diferentes. En el análisis principal, obtuvimos resultados que sugieren que incluso las etiquetas morfosintácticas predichas producidas por etiquetadores muy precisos eran peores que no usar ninguna etiqueta en absoluto, y que las etiquetas morfosintácticas "gold standard" (producidas por expertos humanos) muestran cierta excepcionalidad. También observamos que el corpus más pequeño se benefició de las etiquetas predichas a pesar de que la precisión fuese menor que para la mayoría de los corpus. Usamos estos dos hallazgos (excepcionalidad de las etiquetas "gold standard" e influencia beneficiosa de las etiquetas en corpus más pequeños) para desarrollar dos análisis secundarios. En primer lugar, evaluamos por qué las etiquetas de referencia anotadas por humanos son tan útiles al establecer lo que los analizadores sintácticos inherentemente aprenden sobre las etiquetas morfosintácticas. Descubrimos que los analizadores parecen aprender una cantidad sustancial de información sobre las etiquetas morfosintácticas, y lo que no logran aprender tiene un fuerte solapamiento con lo que los etiquetadores tampoco aprenden. En una serie de experimentos de enmascaramiento, observamos que, de hecho, fueron las situaciones en las que los etiquetadores no pudieron predecir las etiquetas morfosintácticas con precisión las que resultaron más útiles. A continuación, ampliamos el trabajo en corpus más pequeños e investigamos la utilidad de las etiquetas morfosintácticas en entornos de bajos recursos.

Consolidamos los hallazgos del análisis original utilizando datos artificiales de bajos recursos, datos de entornos reales de bajos recursos, y también datos de bajos recursos a los que se aplican técnicas de aumento de datos. Resultó claro que incluso cuando las etiquetas morfosintácticas predichas provenían de etiquetadores que solo lograban una precisión baja, para el análisis de bajos recursos seguían resultando útiles. Fundamentalmente, el trabajo sobre la eficacia de las etiquetas morfosintácticas simplemente resalta la necesidad de evaluar si son beneficiosas para un idioma determinado en un contexto dado, en lugar de dar por sentado que en el peor de los casos no agregarán nada cuando en realidad pueden dañar el rendimiento.

Al principio de este resumen extendido, resumí sucintamente esta tesis formulando dos preguntas. Por lo tanto, parece razonable considerar qué tan bien (si es que lo hace) la tesis logró responderlas. Las preguntas eran:

- P1. ¿Podemos mejorar la eficiencia de los analizadores sintácticos modernos con respecto a la precisión y la velocidad del análisis?
- P2. ¿Podemos explicar al menos parcialmente la variación en el análisis del rendimiento observada en ciertos contextos?

Podemos responder positivamente a ambas. A este respecto, se puede decir que la tesis ha sido un éxito.